Formale Fehlermodellierung für verteilte reaktive Systeme

Max Breitling

Lehrstuhl für Software & Systems Engineering
Institut für Informatik
Technische Universität München

Institut für Informatik
der Technischen Universität München
Lehrstuhl für Software & Systems Engineering

Formale Fehlermodellierung für verteilte reaktive Systeme

Max Dieter Breitling

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Christoph Zenger

Prüfer der Dissertation:

1. Univ.-Prof. Dr. Manfred Broy
2. Univ.-Prof. Dr. Eike Jessen

Die Dissertation wurde am 14.03.2001 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 18.06.2001 angenommen.

Zusammenfassung

An die Entwicklung heutiger informationstechnischer Systeme werden aufgrund ihrer hohen Komplexität, strenger Rahmenbedingungen und vielfältiger geforderter Qualitätsmerkmale hohe Anforderungen gestellt. Insbesondere für Systeme oder Systemteile, deren Mängel zu erheblichen Schäden führen könnten, ist die Entwicklung mit Hilfe formaler Methoden ein Ansatz, diesen Anforderungen zu begegnen. Formale Methoden ermöglichen eine hohe Präzision in Spezifikationen und die akkurate Verifikation kritischer Systemeigenschaften.

Die Verwendung formaler Methoden zielt üblicherweise ab auf den Nachweis der Abwesenheit von Fehlern in Systemen. Da ein formales Modell im allgemeinen aber eine Abstraktion des realen Systems darstellt, die daher potentielle, beispielsweise implementierungsbedingte Fehler nicht repräsentieren kann, beinhaltet Fehlerfreiheit im formalen Modell eine idealisierende Annahme. Diese Idealisierung ist für frühe Entwicklungsphasen zulässig, in denen die eigentliche, intendierte Kernfunktionalität des Systems im Vordergrund der Untersuchungen steht.

Im Rahmen einer weiteren Systementwicklung sollte es aber möglich sein, Fehler als unerwünschten, aber dennoch unvermeidbaren Teil eines Systems explizit aufzunehmen und zu behandeln. In dieser Arbeit wird hierfür eine formale Unterstützung präsentiert.

Die formale Methodik Focus wird dazu um Begriffe und Notationen erweitert, mit denen verschiedenste Klassen von Fehlern eines verteilten, reaktiven Systems sowohl in abstrakter, eigenschaftsorientierter Black-Box Sicht als auch im Kontext von Transitionssystemen als sogenannte Modifikationen explizit dargestellt werden können. Auf der Grundlage einer formalen Darstellung wird es möglich, präzise und nachweisbare Aussagen über Fehlertoleranz eines Systems und die Auswirkungen von Fehlern zu machen. Zur Unterstützung einer Systementwicklung wird die schrittweise, auch nachträgliche Einführung von Techniken zur Fehlererkennung, zur Generierung von Fehlermeldungen und zur Fehlerkorrektur in ein bestehendes System auf systematische Weise ermöglicht. Die Praktikabilität dieser Techniken wird durch begleitende Beispiele illustriert.

Damit wird ein weiterer Schritt hin zu einer durchgängigen und praktischen Verwendbarkeit formaler Methoden geleistet.

Danksagung

Meine Arbeit ist nur mit der vielfältigen Unterstützung meiner Kollegen, Freunde und Familie möglich geworden. Dafür möchte ich mich an dieser Stelle ganz herzlich bedanken!

Insbesondere möchte ich mich bei Prof. Manfred Broy bedanken für seine fachliche wie organisatorische Betreuung und Unterstützung während aller Entstehungsphasen dieser Arbeit. Für die Übernahme des Zweitgutachtens und seine Hinweise zum Fehlerbegriff bin ich Prof. Eike Jessen sehr dankbar.

Großen Dank schulde ich Jan Philipps. Unsere gemeinsame intensive Projektarbeit hat nicht nur zu einer guten Fundierung meiner Arbeit geführt, sondern hat mir auch sehr viel Spaß gemacht. Ingolf Krüger danke ich für seine vielfältige konstruktive Unterstützung, unter anderem durch zahlreiche fachliche Diskussionen und Anregungen. Katharina Spies und Bernhard Schätz gebührt mein Dank für ihre motivierenden Hilfestellungen und Ratschläge, insbesondere in den frühen und konzeptionellen Phasen der Arbeit.

Nicht zuletzt bin ich sehr dankbar für die viele Geduld, Nachsicht und Aufmunterung, die Konstanze, meine Freunde und meine Familie während der Fertigstellung dieser Arbeit für mich aufbrachten.

Für die Durchsicht einer Vorversion der Arbeit und ihre konstruktiven Vorschläge möchte ich Prof. Manfred Broy, Prof. Eike Jessen, Katharina Spies, Konstanze Tauber und Ingolf Krüger meinen herzlichsten Dank ausdrücken.

Die Arbeit wurde finanziell durch die Deutsche Forschungsgemeinschaft DFG im Rahmen des Sonderforschungsbereichs 342 unterstützt.

Inhaltsverzeichnis

1 Einführung **1**
1.1 Einleitung . . . 1
1.2 Umfeld . . . 3
1.3 Motivation . . . 5
1.4 Ziel . . . 6
1.5 Überblick . . . 7

2 Der Fehlerbegriff **9**
2.1 Fehler . . . 9
2.1.1 Fehlerbegriffe der Literatur . . . 11
2.1.2 Klassifikationen . . . 13
2.1.3 Fehler als Soll/Ist-Abweichung . . . 17
2.1.4 Umgang mit Fehlern . . . 21
2.2 Fehlertoleranz . . . 24
2.2.1 Formale Ansätze . . . 25
2.2.2 Techniken der Fehlertoleranz . . . 29
2.3 Zusammenfassung . . . 35

3 Formales Systemmodell **37**
3.1 Überblick . . . 37
3.2 Mathematische Grundlagen . . . 38
3.3 Systemschnittstelle . . . 40
3.4 Systemverhalten . . . 41
3.5 Black-Box Spezifikationen . . . 44
3.6 Zustandstransitionssysteme . . . 48
3.6.1 Graphische Darstellung . . . 53
3.6.2 Tabellarische Darstellung . . . 54
3.7 Komposition . . . 55
3.7.1 Komposition von Black-Box Spezifikationen . . . 57
3.7.2 Komposition von Zustandsmaschinen . . . 58
3.8 Verifikation von Eigenschaften . . . 59
3.8.1 Eigenschaften von Transitionssystemen . . . 59
3.8.2 Verifikationsdiagramme . . . 62
3.8.3 Black-Box Eigenschaften von Transitionssystemen . . . 66
3.9 Verfeinerungen . . . 67

3.9.1 Verhaltensverfeinerung 67
3.9.2 Schnittstellenverfeinerung 68
3.9.3 Kompositionalität 69
3.10 Entwicklungsprozeß 69
3.11 Zusammenfassung 71

4 Formale Modellierung von Fehlern 75
4.1 Modifikation der Schnittstelle 76
4.1.1 Erweiterung der Schnittstelle 77
4.1.2 Einschränkung der Schnittstelle 81
4.2 Modifikation des Verhaltens 82
4.2.1 Modifikationen von Relationen 83
4.2.2 Modifikationen von Black-Box Spezifikationen 85
4.2.3 Modifikationen von Transitionssystemen 86
4.2.4 Modifikationskomponenten 89
4.3 Formalisierung der Fehlerbegriffe 92
4.3.1 Fehler 93
4.3.2 Fehlzustand 93
4.3.3 Versagen 95
4.4 Fehlertoleranz 98
4.4.1 Fehlertoleranz als korrespondierende Modifikation 98
4.4.2 Fehlertoleranz als Dämpfung (quantitativ) 100
4.4.3 Fehlertoleranz als Dämpfung (qualitativ) 101
4.5 Eigenschaften von Modifikationen 103
4.5.1 Kombination von Modifikationen 103
4.5.2 Fortpflanzung von Modifikationen 106
4.5.3 Korrespondierende Modifikationen 110
4.6 Fehler in der Systemumgebung 112
4.6.1 Explizite Umgebungsannahmen 113
4.6.2 Implizite Umgebungsannahmen 115
4.7 Zusammenfassung und Diskussion 119

5 Methodischer Umgang mit Fehlern 121
5.1 Beispiele für Fehlermodelle 121
5.1.1 Crash failure 122
5.1.2 Fail-stop failure 123
5.1.3 Omission failure 123
5.1.4 Byzantine failure 124
5.2 Formulierung von Fehlerannahmen 125
5.3 Fehlererkennung 127
5.4 Einführung von Fehlermeldungen 130
5.5 Fehlerkorrektur 132
5.5.1 Ergänzung von korrigierenden Transitionen 133
5.5.2 Treiberkomponenten 137
5.6 Erhöhung der Systemrobustheit 139
5.6.1 Explizite Umgebungsannahmen 139

5.6.2 Implizite Umgebungsannahmen 141
5.7 Wiederverwendung von Beweisen 143
5.7.1 Invariantendiagramme 144
5.7.2 Fortschrittsdiagramme 148
5.8 Zusammenfassung und Diskussion 153

6 Zusammenfassung und Ausblick 157
6.1 Beitrag dieser Arbeit 157
6.2 Weiterführende Themengebiete 161

Literaturverzeichnis 165

Abbildungsverzeichnis 171

Kapitel 1

Einführung

Nach der folgenden Einleitung, die die Herausforderungen der heutigen Informatik illustriert, erläutert dieses Kapitel in Abschnitt 1.2 das Umfeld der Arbeit und präzisiert damit ihren Titel. Im Anschluß an die Motivation in Abschnitt 1.3 formulieren wir in Abschnitt 1.4 die Ziele der Arbeit und geben einen kurzen Überblick über die Inhalte der folgenden Kapitel.

1.1 Einleitung

Durch die rasche Entwicklung der Leistung informationstechnischer Systeme und ihre damit verbundene rasante Ausbreitung in immer mehr Anwendungsbereiche sind die Qualitätsanforderungen an diese Systeme enorm gewachsen. Rechner spielen eine zentrale Rolle bei der Steuerung der vielfältigsten Systeme wie Flugzeuge, Autos, Raketen, Haushaltsgeräte, Telefone, Waffensysteme, Fernseher und andere Unterhaltungsgeräte, Atomkraftwerke, Kinderspielzeuge, die Strom-, Gas- und Wasserversorgung, alle Arten von Telekommunikation; sie bewältigen den für die Mehrzahl der finanziellen Transaktionen notwendigen Informationsfluß und sind in Büros mittlerweile unverzichtbar. Sogar die Entwicklung neuer Systeme ist nur noch mit bereits bestehenden Systemen möglich.

Durch ständige technische Neuerungen wächst die Zahl der Möglichkeiten und damit der Anwendungsgebiete von IT-Systemen stetig. Ihre Verwendung reicht immer tiefer in alle Anwendungsbereiche hinein, was die Komplexität der Systeme weiter erhöht. Alle Systeme und Dienste werden idealerweise miteinander gekoppelt, und was technisch möglich ist, soll möglichst schnell auch realisiert werden. Der Markterfolg neuer Produkte und Dienstleistungen hängt oft von der Geschwindigkeit ab, in der sie verfügbar gemacht werden.

Mit zunehmender Verwendung und Vernetzung der IT-Systeme in allen Bereichen des Lebens wächst die Abhängigkeit von ihrer Verfügbarkeit, Zuverlässigkeit, Sicherheit und Korrektheit stark an. Ein auftretendes Versagen kann in immer komplexer werdenden Systemen fatale Folgen haben. Da auch die Zahl von Anwendungen steigt, wachsen auch die Fehlerwahrscheinlichkeiten: Auch ein unwahrscheinlicher Fehler wird bei millionenfacher Verwendungen von Geräten irgendwann doch auftreten.

Die gestellten Kriterien an die Qualität betreffen im allgemeinen eine ganze Reihe von Aspekten, die je nach Anwendungsgebiet unterschiedlich gewichtet sein können. So sollen Systeme *Verläßlichkeit* aufweisen, also die von ihnen erwartete Leistung auf *zuverlässige* Weise erbringen. Ein System soll *sicher* (im Sinne von *safe*) sein, so dass keinerlei Gefährdung von ihm ausgeht und es nicht möglicherweise Menschenleben bedroht oder Schäden anderer Art verursachen kann. Es soll auch *sicher* (im Sinne von *secure*) sein, also keine Informationen preisgeben an Personen, die nicht berechtigt sind, diese zu erhalten. Ein System soll *korrekt* sein, also genau das tun, was von ihm erwartet wird. Ausführlichere Diskussionen verschiedener Qualitätsaspekte finden sich beispielsweise in [58, 45, 43].

Es ergibt sich noch eine Reihe weiterer Fragen, wie beispielsweise die Erwartungen an ein System definiert werden, wie die Zuverlässigkeit ermittelt und bewertet wird, wie eine adäquate Spezifikation erstellt wird, die selbst wieder als *korrekt* bezeichnet werden kann, und viele mehr. Diesen Fragestellungen wird in der Disziplin des *Requirements Engineering* nachgegangen. Wir gehen im Rahmen dieser Arbeit von dem vereinfachten Ansatz aus, dass bereits eine Spezifikation vorliegt, die *alle* relevanten Qualitätsaspekte umfaßt, die wir von einem System fordern. Erfüllt ein System also seine Spezifikation, müssen keine weiteren Kriterien überprüft werden.

Doch auch unter der Voraussetzung, dass alle Anforderungen an ein System im Rahmen einer Spezifikation bereits definiert sind, ist es eine große Herausforderung, eine entsprechende Implementierung zu entwickeln. Dies ist im wesentlichen in der hohen Komplexität der Systeme begründet. Sie bestehen aus einem diffizilen, ausbalancierten Zusammenspiel vieler Komponenten in Hard- und Software, die untereinander und mit anderen Systemen interagieren. Es werden meist bestehende Altsysteme oder Systemteile integriert, deren Funktionsweise unter Umständen nur teilweise dokumentiert ist. Systeme können so groß und komplex werden, dass niemand sie mehr zugleich vollständig und im Detail verstehen kann. Bei der Durchführung realer Projekte kommt hinzu, dass sich während der Entwicklung Ziele, Spezifikationen, die Systemumgebung und zugrundeliegende Hardware oder Betriebssysteme ändern und nur knappe Ressourcen von Zeit und Geld zur Verfügung stehen.

Die Informatik hat in den letzten Jahren, beispielsweise durch Beschreibungstechniken, Vorgehensmodelle, (domänenspezifische) Systemarchitekturen oder Entwicklungswerkzeuge bereits einige Fortschritte gemacht, befindet sich aber im Vergleich zu anderen Ingenieurdisziplinen, die über jahrzehntelange Erfahrungen verfügen, noch immer in ihren Anfängen. Die in den späten Sechziger Jahren formulierte *Software-Krise* brachte die Disziplin des *Software-Engineerings* hervor, die zum Ziel hat, wissenschaftlich fundierte Methoden und Techniken bereitzustellen, mit denen den genannten Problemen begegnet werden kann.

Diese Arbeit leistet den Beitrag, die vielfältigen Methoden des Software-Engineerings um den speziellen Aspekt der Fehlermodellierung und -behandlung zu ergänzen und damit die Palette der Techniken zur Erstellung hochqualitativer Systeme zu erweitern. Dazu wird der Umgang mit *Fehlern* in *verteilten reaktiven Systemen* im Rahmen einer formalen Methodik aus systematische Weise ermöglicht. Wir werden dazu im folgenden das Umfeld und die Ziele der Arbeit genauer motivieren und darstellen.

1.2 Umfeld

Das thematische Umfeld der Arbeit ergibt sich unmittelbar aus den Stichworten des Titels. Wir wollen unsere Ansätze auf eine formal fundierte Grundlage stellen und im folgenden daher die wesentlichen Kennzeichen formaler Methoden kurz charakterisieren. Wir konzentrieren uns im wesentlich auf die Systemklasse der verteilten, reaktiven Systeme und wollen diese Wahl begründen. Dem Begriff des Fehlers und der Thematik der Fehlermodellierung ist mit Kapitel 2 ein eigenes Kapitel gewidmet.

Formale Methoden Der Begriff der *formalen Methoden* umfaßt eine Palette von Methoden und Hilfsmitteln, die eine Systementwicklung auf Basis mathematischer Grundlagen auf eine präzise und verifizierbare Weise unterstützt.

Im Idealfall stellen die formalen Methoden dazu Notationen und eine Sprache bereit, mit der es möglich wird, Spezifikationen mit einer klar definierten Semantik zu formulieren. Im Gegensatz zu informellen Spezifikationen wird es damit eher möglich, die Forderungen nach Eindeutigkeit und Konsistenz zu erfüllen. Eigenschaften einer Spezifikation oder eines Systems und durch sie implizierte Konsequenzen können unter Verwendung eines logischen Kalküls formal bewiesen werden. Entwicklungsschritte, die von Spezifikationen als abstrakten Systembeschreibungen hin zu einer ausführbaren Implementierung führen, werden durch geeignete Verfeinerungsbegriffe unterstützt. Werkzeuge stellen dazu die notwendige Infrastruktur zur Bearbeitung von Systembeschreibungen und eine möglichst automatisierte Durchführung von Beweisen bereit.

Als Beispiele für formale Ansätze lassen sich nennen: die Methoden RAISE, Unity, VDM, Z, TLA, die B-Methode, algebraische Ansätze wie CASL, Focus mit dem Werkzeug AutoFocus und Petri-Netze. Diese Beispiele bieten vielmals nur Teilaspekte formaler Methoden, und stellen oft nur eine Sprache zur Spezifikation bereit – ohne entsprechende Konzepte zur Verfeinerung und ohne Werkzeuge. Sie unterscheiden sich teilweise wesentlich in ihrem zugrundeliegenden Systemmodell, das unter anderem die prinzipiellen Paradigma für die Kommunikation, die Ausführungsmodelle und die Modellierung von Zeit festlegt.

Die praktische Verwendbarkeit formaler Methoden ist noch immer Gegenstand von Diskussionen. Auch wenn ihr Nutzen nicht immer eindeutig nachgewiesen werden kann, werden sie in zunehmenden Maße erfolgreich eingesetzt, vor allem in sicherheitskritischen Anwendungsbereichen. Durch die in Abschnitt 1.1 genannten wachsenden Anforderungen an die Systeme und durch ständige Fortschritte auf dem Gebiet der formalen Methoden wird ihre Bedeutung und Einsetzbarkeit weiter zunehmen. In [26, 68] finden sich sowohl viele Beispiele als auch Literaturhinweise und Erfahrungsberichte für verschiedenste formale Methoden und ihre Anwendungen.

Der Ansatz dieser Arbeit zur Fehlermodellierung basiert auf der formalen Methodik Focus, deren relevante Aspekte wir im Detail in Kapitel 3 vorstellen werden.

Verteilte, reaktive Systeme Die Systemklasse der reaktiven Systeme zeichnet sich, wir ihr Name schon sagt, durch eine beständige *Reaktivität* aus, und unterscheidet sich damit von sogenannten *transformationellen* Systemen. Die transformationellen Systeme

werden zusammen mit einer Eingabe gestartet, berechnen ein Ergebnis und terminieren. Ein Neustart mit einer weiteren Eingabe ist im allgemeinen möglich, aber unabhängig vom vorherigen Lauf; ein errechnetes Resultat hängt nur von der Eingabe ab, die von Anfang an vollständig zu Verfügung steht. Transformationelle System kann man charakterisieren als Funktionen, die eine Ein- auf eine Ausgabe abbilden. So gehören klassische sequentielle Programme ohne Interaktion mit ihrer Umgebung in diese Klasse.

Ein reaktives System arbeitet dagegen eher kontinuierlich: Es empfängt stetig die Nachrichten, die von der Umgebung in möglicherweise willkürlicher Folge und Häufung gesendet werden, und darauf reagiert im allgemeinen mit Ausgaben. Das System terminiert nicht, sondern bleibt reaktiv. Abläufe sind also potenziell unendlich. Die Reaktion eines Systems kann dabei von aktuellen Eingaben und der Vorgeschichte, also vorherigen Eingaben und Entscheidungen des Systems abhängen. Sie sind demzufolge allgemeiner und deutlich komplexer als transformationelle Systeme.

Reaktive Systeme stellen eine große und wichtige Systemklasse dar, die oft auch in sicherheitskritischen Bereichen ihre Anwendung findet – mit den damit verbundenen hohen Anforderungen an Korrektheit und beständige Verfügbarkeit. Beispiele sind Betriebssysteme, Systeme zur Nachrichtenübermittlung wie Basisstationen von Mobilfunksystemen oder Netzknotenrechner sowie eingebettete Systeme zur Steuerung von Fahrzeugen, Küchengeräten oder industriellen Herstellungsprozessen.

Ein *verteiltes* System besteht nicht aus einer einzigen Einheit, sondern aus mehreren Komponenten, die miteinander interagieren und so gemeinsam eine Gesamtleistung erbringen. Gründe für die Verteilung eines Systems können physikalische oder effizienzbedingte Notwendigkeiten sein, und ebenso ein Mittel, die Komplexität eines Systems durch eine Aufteilung in überschaubarere Teile handhabbar zu machen. Beispielsweise muss ein internationales Buchungssystem auf viele physikalische Standorte verteilt sein, und wird sinnvollerweise in logische Anwendungsmodule strukturiert.

Die verteilten, reaktiven Systeme stellen also eine Systemklasse dar, die in der Praxis eine relevante Rolle spielt. Die Entwicklung dieser Systeme stellt durch ihre inhärente Komplexität eine große Herausforderung dar, so dass eine formale Unterstützung notwendig oder zumindest lohnenswert ist. Das in Kapitel 3 vorgestellte Systemmodell ist für diese Klasse hervorragend geeignet.

Fehler Fehler spielen in sehr vielfältiger Weise eine Rolle im Umfeld informationstechnischer Systeme: Sie können in Erwartungen, Spezifikationen, Programmen, Teilkomponenten, lokal oder verteilt, in Hard- oder Software oder auch in Abläufen auftreten. Sie sind unter Umständen nicht exakt lokalisierbar und nicht quantifizierbar. Sie können einen statischen oder dynamischen Charakter haben. Mit Fehlern kann auf verschiedenste Weise umgegangen werden: So können sie vermieden, ignoriert, erkannt, behandelt, gesucht, entfernt, gemeldet und toleriert werden.

Die vielfältigen Interpretationen des Fehlerbegriffes haben jedoch eine Gemeinsamkeit: Ein Fehler ist immer nur relativ zu einer definierten Korrektheit zu verstehen, ist also zu identifizieren mit einer Abweichung eines (vorliegenden) *Ist* von einem (erwünschten) *Soll*. Eine intensivere Diskussion, Charakterisierung und Klassifizierung des Begriffes ist (als Grundlage dieser Arbeit) lohnenswert und ist in Kapitel 2 zu finden.

1.3 Motivation

Der Begriff des *Fehlers* spielt in einer Systementwicklung, die mit Hilfe verbreiteter formaler Hilfsmittel durchgeführt wird, kaum eine Rolle. Es mag sogar widersprüchlich erscheinen, hier einen Zusammenhang zu sehen: Formale Methoden werden ja gerade verwendet, um korrekte, also *fehlerfreie* Systeme zu entwickeln. Diese Sicht ist aber nur zutreffend, wenn man *Fehler* im Sinne von *Fehlern im Entwicklungsprozeß* versteht, wie beispielsweise nicht aufgedeckte Inkonsistenzen in Spezifikationen oder inkorrekte Verfeinerungsschritte. Die Verwendung formaler Methoden dient hier tatsächlich der *Fehlervermeidung*.

Eine andere Klasse von Fehlern sind aber Fehler, die in einer abstrakten Spezifikation nicht auftreten, in einer realen Implementierung aber durchaus. Eine Abstraktion ist in der Regel mit idealisierenden Annahmen verbunden, die beispielsweise Laufzeitfehler nicht berücksichtigen, die durch Mängel in der Hardware oder in verwendeten Komponenten ausgelöst werden und zum Versagen des Systems führen können. Diese Abstraktion von beispielsweise physikalischen Defiziten ist selbstverständlich erwünscht, denn sie trägt zur Reduktion der Komplexität bei der Systementwicklung bei, da sie ein System auf seine für die intendierte Funktionalität relevanten Aspekte reduziert. Formale Methoden betrachten diese Klasse von Fehlern meist nicht, sondern gehen implizit davon aus, dass diese Fehler auf implementierungsnaher Ebene abgefangen und behandelt werden, also außerhalb des formalen Rahmens. Damit wird die Anwendbarkeit der formalen Methoden auf einen Teil der Systementwicklung beschränkt.

Ein Ziel der Arbeiten auf dem Gebiet der formalen Methoden ist es aber, die ermöglichte Präzision nicht nur auf Ausschnitte eines Systems zu begrenzen, sondern eine durchgängige Verwendung zu ermöglichen, beispielsweise durch einen automatisierten Übergang von Spezifikationen zu implementierungsnahen Systembeschreibungen bis zur Generierung von Code. Dies erfordert den Umgang mit den letztendlich unvermeidbaren Fehlern auch innerhalb der abstrakten Formalismen. Die durch die Abstraktion induzierte, idealisierte Annahme von Fehlerfreiheit soll bewußt und auf systematische Weise aufgegeben werden können. Fehler können damit formal erfaßt werden.

Eine weitere Motivation zur Integration von Fehlern ist eine wünschenswerte Unterstützung der Entwicklung sogenannter *fehlertoleranter Systeme*. In diesen spielen Fehler eine prominente Rolle, da Aussagen über das Systemverhalten diese explizit referenzieren. Fehlertoleranz ist eine Eigenschaft, die gerade bei sehr hohen Sicherheits- und Zuverlässigkeitsforderungen an Systeme wichtig ist, für die sich die Verwendung formaler Methoden also empfiehlt.

Der Mangel klarer Begriffe im Umfeld der Fehler und einer Systematik im Umgang mit ihnen wird auch von anderen Autoren identifiziert. Felix C. Gärtner hält die häufig verwendeten Begriffe wie *fault*, *error* und *failure* für unscharf definiert [28]

> *... These terms are admitttedly vague, and here again formalization can help clarification.* (Seite 4)

Flaviu Cristian fordert in [23] eine einheitliche Begriffsbildung für die Fehlermodellierung:

> *One has to stay in control of not only the standard system activities when all components are well, but also of the complex situations which can occur when some components fail. The difficulty of this task can be exacerbated by the lack of clear structuring concepts and the use of a confusing terminology. Presently, it is quite common to see different people use different names for the same concept or use the same term for different concepts. For example, what one person calls a failure, a second person calls a fault, and a third person might call it an error.*

Jeanette Wing beschreibt in [70] die folgenden Forderungen von Software-Ingenieuren hinsichtlich der Ausdrucksmöglichkeiten in Spezifikationen:

> *When I have asked the question* [“Why specify formally?“] *of a software engineer, here are the kinds of responses I have heard: ...*
>
> - *to specify what happens if an error occurs*
> - *to specify the right thing happens if an error occurs*
> - *to make sure this error never occurs ...* (Seite 3)

Ian Sommerville fordert in [62], die möglichen Versagen explizit in Spezifikationen aufzunehmen um damit mit diesen umgehen zu können:

> *When writing a reliability specification, the specifier should identify different types of failure and consider whether these should be treated differently in the specification.* (Seite 357)

Lee und Anderson fordern eine Beschreibungstechnik, in der das normale Verhalten vom Sonder- bzw. Fehlerverhalten getrennt gehalten wird [44]:

> *... it is highly desirable that a framework is adopted in which the normal activity of the system is clearly distinguished from its abnormal (i.e. fault tolerance) activity ...* (Seite 245)

Die Defizite, die sich sowohl durch die unklaren Begriffe im Umfeld von Fehlern ergeben als auch durch den unsystematischen Umgang mit Fehlern im Rahmen formaler Methoden, sind also klar erkennbar. Diese Arbeit hat sich zum Ziel gesetzt, diese Defizite zu reduzieren.

1.4 Ziel

Das Ziel der Arbeit, das durch die beschriebenen Defizite motiviert wurde, wird nun explizit und zusammenfassend formuliert:

> *Die formale Methodik* FOCUS *wird erweitert um Begriffe und Vorgehensweisen, die einen expliziten Umgang mit verschiedenartigsten Fehlern, ihren Eigenschaften und Auswirkungen im Rahmen einer Systementwicklung*

> *für verteilte reaktive Systeme mit hohen Qualitätsanforderungen unterstützt. Systeme und ihre potentiellen Fehler werden damit auf systematische Weise modellierbar, analysierbar und behandelbar.*

Im einzelnen erfordert dieses Ziel die Lösung der folgenden Teilaufgaben:

- Zunächst soll der *Fehlerbegriff* im Rahmen eines formalen Systemmodells untersucht und möglichst in Übereinstimmung mit etablierten, informellen Charakterisierungen des Begriffs formal definiert werden.
- Die *Eigenschaften* des Fehlerbegriffes sind zu untersuchen und zu differenzieren. Fragen nach der Auswirkung von Fehlern in Komponenten auf das Verhalten des Gesamtsystems, nach der Wirkung von Kombinationen und Überlagerungen von auftretenden Fehlern und nach den Zusammenhängen von internen und externen Fehlern sollen unter Ausnutzung eines formalen Kalküls beantwortbar werden.
- Um mit konkreten Fehlern in einem konkreten System umgehen zu können, müssen angemessene *Beschreibungstechniken* für Fehler angegeben werden, die es erlauben, auf verschiedenen Abstraktionsebenen die durch Fehler bedingten Abweichungen von Ist und Soll zu formulieren.
- Fehler in einem System erfordern die Integration von Techniken, die mit diesen auf sinnvolle Weise umgehen. Dazu sind *methodische* Anleitungen zu erarbeiten, wie ein idealisiertes, fehlerfreies System schrittweise zu einem fehlerbehafteten System entwickelt werden kann, in dem Fehlern durch Fehlererkennung, -behebung und -toleranz begegnet wird.
- Der *Nutzen* der definierten Begriffe und des vorgeschlagenen methodischen Umgangs sollte durch Fallbeispiele nachgewiesen werden.

Die angestrebte Erweiterung formaler Methoden stellt einen weiteren Schritt hin zu einer durchgängigen Verwendung formaler Methoden dar, die nicht nur Fehlervermeidung, sondern den expliziten, bewußten und präzisen Umgang mit Fehlern ermöglicht.

1.5 Überblick

In Kapitel 2 charakterisieren und diskutieren wir den Begriff des *Fehlers* auf Grundlage relevanter Literatur. Wir differenzieren Fehler in Fehlerursachen, Fehlerzustände und Versagen, geben eine Klassifikation der Begriffe an und beschreiben Techniken zum Umgang mit Fehlern. Das Thema der *Fehlertoleranz* wird ausführlicher behandelt, indem sowohl bekannte formale Ansätze vorgestellt werden als auch etablierte Techniken, die in fehlertoleranten Systemen zum Einsatz kommen.

Um die Fehlerbegriffe formal definieren zu können, ist eine mathematische Grundlage notwendig. Dazu präsentieren wir in Kapitel 3 ein Systemmodell, das zur Modellierung verteilter, reaktiver Systeme geeignet ist. Ein System wird darin als eine Menge interagierender Komponenten dargestellt, die über Kommunikationskanäle asynchron Nachrichten austauschen. Die Kommunikationsgeschichte wird durch Nachrichtenströme,

Systemverhalten durch Relationen von Nachrichtenströmen dargestellt. Es werden zwei Beschreibungstechniken für Systeme präsentiert, die sich für Systembeschreibungen auf verschiedenen Abstraktionsebenen eignen. Der Übergang zwischen den Ebenen basiert auf einer Verifikationtechnik, deren Beweise mit Beweisdiagrammen visualisiert werden können. Schliesslich beschreiben wir das Konzept der Verfeinerung und einen idealisierten Entwicklungsprozeß. Dieses Systemmodell sieht die explizite Modellierung von Fehlern noch nicht vor.

In Kapitel 4 erweitern wir das Systemmodell um sogenannte *Modifikationen* $\mathcal{M}$. Mit diesen ist es möglich, eine Veränderung sowohl von Schnittstellen als auch von Systemverhalten zu beschreiben, darstellbar mit Hilfe beider Beschreibungstechniken. Für ein System S beschreibt $S \triangle \mathcal{M}$ das veränderte System. Damit steht ein geeigneter Formalismus bereit, mit dem Fehler (als Abweichung eines Ist von einem Soll) und verwandte Begriffe formal definiert werden. Darüber hinaus werden in diesem Kapitel Kombinationen von Modifikationen und ihre Eigenschaften untersucht, und die Modellierung von externen Fehlern dargestellt.

Um Nutzen aus einer formalen Methodik ziehen zu können, ist neben einer Sprache zur Darstellung von Systemen und ihren möglichen Fehlern auch ein zweckmäßiger, zielgerichteter Umgang erforderlich. In Kapitel 5 werden methodische Hinweise zum Umgang mit Fehlern gegeben. Wir erläutern, wie präzise Fehlermodelle auch von zusammengesetzten Systemen formuliert und mit welchen Techniken Fehler im Rahmen unseres Fehlermodells erkannt, gemeldet und korrigiert werden können. Es wird angegeben, wie die Robustheit von Systemen gegenüber externen Fehlern erhöht und wie aus Beweisen zu fehlerfreien Systemen Beweise für fehlerbehaftete Systeme abgeleitet werden können.

In allen Kapiteln verdeutlichen wir alle wesentlichen Konzepte mit Hilfe vieler Beispiele. Dabei konzentrieren wir uns auf die durchgängige Verwendung zweier einfacher Systeme *Merge* und *Buffer*, die trotz ihrer Einfachheit eine ausreichende Komplexität aufweisen, um die jeweils relevanten Aspekte demonstrieren zu können.

Schliesslich werden in Kapitel 6 die Beiträge dieser Arbeit zusammengefaßt und verbleibende sowie neu aufgeworfene, offene Probleme identifiziert.

Kapitel 2

Der Fehlerbegriff

Als Grundlage für einen formalen, präzisen Umgang mit Fehlern wollen wir den facettenreiche Begriff des *Fehlers* mit seinen verschiedenen Bedeutungen zunächst informell diskutieren. Wir fassen in diesem Kapitel verschiedene Begriffe zusammen, die sich dazu in der Literatur aus dem Bereich der Informationstechnik finden lassen.

Dazu beschreiben wir die Differenzierung von Fehlern in Fehler*ursache*, Fehler*zustand* und *Versagen* mit ihren Zusammenhängen, und geben Klassifizierungen dieser Begriffe an. Nach Angabe verschiedener Möglichkeiten im Umgang mit Fehlern fokussieren wir auf die Fehlertoleranz, welche wir zunächst informell beschreiben und darauf entsprechende formale Ansätze diskutieren. Schließlich präsentieren wir prinzipielle, etablierte Techniken der Fehlertoleranz.

2.1 Fehler

Der Begriff des Fehlers ist sehr unterschiedlich interpretierbar. Wir wollen anhand verschiedener Beispiele den sehr unterschiedlichen Charakter verschiedener Arten von Fehlern illustrieren. In den meisten Fällen wird dabei der Begriff des *Fehlers* verwendet.

Bei der Nutzung eines Diskettenlaufwerks kann ein Rechner einen Lesefehler melden. Eine Systemspezifikation kann einen Fehler aufweisen, der in einem Review-Prozeß aufgedeckt wird. Es können falsche Informationen in einem Benutzerhandbuch enthalten sein, die Benutzer zu einer falschen Bedienung oder zu falschen Erwartungen gegenüber eines Systems führen. Die Ausführung eines Programmes kann unerwartet nicht terminieren. Das Layout eines mit einem Textverarbeitungsprogramms erstellten Dokumentes kann im Ausdruck anders aussehen als erwartet. Der Kontostand auf einem Kontoauszug kann überraschend niedrig sein, so dass er als fehlerhaft empfunden wird. Ein Steuersystem kann versagen, so dass sogar Menschenleben gefährdet werden, beispielsweise durch unerwartet ausgelöste oder nicht oder zu spät ausgelöste Airbags.

Diese Beispiele zeigen, wie unterschiedlich die Ursachen, die Auswirkungen, die Art und Weise der Feststellung, die Wahrnehmung, die Lokalisierung und Quantifizierung von sogenannten *Fehlern* sein können.

Während ein Fehler beim Lesen eines Diskettenlaufwerks unter Umständen vom Rechner selbst erkannt und an einen Benutzer weitergegeben wird, und eine klare Ursache dafür gefunden werden kann, ist ein Fehler in einer Spezifikation vielleicht nur schwer zu erkennen und zu lokalisieren; seine Ursachen können in Mängeln eines Entwicklungsprozesses liegen, und die Folgen können sowohl wieder Fehler im Benutzerhandbuch sein als auch unerwartetes, vielleicht sogar unsicheres oder unzuverlässiges Systemverhalten.

Die Vielfalt von Bedeutungen, mit denen „Fehler“ in Verbindung gebracht wird, legt es nahe, diesen Begriff hier durch eine genauere Differenzierung zu diskutieren.

Die obigen Beispiele zeigen eine Gemeinsamkeit, die alle Arten von Fehlern aufweisen: Ein Fehler beinhaltet immer *eine Abweichung eines Ist von einem Soll.* Fehler lassen sich also nur relativ definieren zu einem Begriff von Korrektheit, der Fehlerfreiheit bedeutet. Diese Korrektheit ist oft mittels einer Spezifikation postuliert. Dennoch können in einer realen Spezifikation natürlich Fehler enthalten sein, wenn die eigentlichen, wahren Anforderungen an ein System nicht korrekt wiedergegeben werden konnten. Die Diskrepanz zwischen einem Ist und einem Soll kann auch durch ein inkorrektes Soll ausgelöst sein, wie das Beispiel des vermeintlich falschen Kontoauszuges zeigt.

Beispiel 2.1 Um die Schwierigkeiten der Qualifizierung, Quantifizierung und Lokalisierung von Fehlern zu verdeutlichen, wollen wir ein einfaches Beispiel angeben. Gefordert sei ein Programm, das für natürliche Zahlen $n \geq 2$ die Fakultät berechnet. Das *Soll* ist damit informell definiert. Betrachten wir nun das folgende, funktionale Programm (ausgedrückt in ML [52]):

```
fun fac (n : int) =
  if n=1 then 2
         else n*fac(n-1);
```

Dieses Programm berechnet nicht das geforderte Ergebnis. Man spricht intuitiv von einem *Fehler* im Programm, ohne sich dabei unbedingt der verschiedenen möglichen Bedeutungen des Begriffes bewußt zu sein. Aber wo liegt der Fehler? Der Fehler läßt sich nicht eindeutig lokalisieren, denn es gibt zwei Möglichkeiten zur Korrektur: Das Resultat im Terminierungsfall ließe sich durch 1 ersetzen (denn $1! = 1$), aber auch die 1 im Konditional der `if`-Anweisung könnte durch eine 2 ersetzt werden, da $2! = 2$ und die Berechnung für $n = 1$ in der Spezifikation ausgeschlossen wurde.

Das Problem der Lokalisierung wird noch deutlicher in folgender „Lösung“ der Aufgabe:

```
fun fac (n : int) =
  if n <= 1 then 1
            else fac(n-1) + fac(n-2);
```

Dieses Programm berechnet eine vollständige andere Funktion. Damit läßt sich nicht einmal auf sinnvolle Weise sagen, *wieviele* Fehler dieses Programm enthält. Es gibt eine unmittelbare Analogie zur natürlichen Sprache: Ist die Aussage eines Satzes nicht korrekt, ist dies im allgemeinen nicht auf ein einzelnes Zeichen oder Wort in diesem Satz zurückzuführen. □

In den folgenden Abschnitten beschäftigen wir uns vornehmlich mit Fehlerbegriffen aus dem informationstechnischen Umfeld, und beschreiben darin weitgehend akzeptierte, in der Literatur häufiger wiederkehrende Darstellungen des Begriffes.

2.1.1 Fehlerbegriffe der Literatur

In der Literatur findet sich eine ganze Reihe von Begriffen aus dem Umfeld der Fehler. Während man in deutschen Texten beispielsweise die Begriffe *Fehler, Versagen, Ausfall, Störung* und *Fehlfunktion* findet, liest man in englischsprachiger Literatur von *faults, errors, failures, malfunctions* und *mistakes*. Zwischen dieses Begriffen lassen sich nicht unmittelbar Entsprechungen finden, da sie sich in ihren intendierten Bedeutungen teilweise überschneiden. So werden die Begriffe *faults* und *errors* meist jeweils beide mit *Fehler* übersetzt, während *failure* im Deutschen differenziert wird in *Ausfall* und *Versagen*. Wir werden die unterschiedlichen Begriffe im folgenden herausarbeiten und orientieren uns dabei an den im wesentlichen übereinstimmenden Charakterisierungen aus [39, 43, 44].

Wir konzentrieren uns auf drei allgemeine, zentrale Begriffe, die im Englischen als *fault*, *error* und *failure* bezeichnet werden. Die verbleibenden Begrifflichkeiten wie *malfunction, mistake, Störung* und *Ausfall* werden wir als spezielle Ausprägungen dieser drei Begriffe interpretieren.

Wir werden im Deutschen (entsprechend der in [43] vorgeschlagenen Übersetzung) „failure" durch *Versagen* und „error" durch *Fehlerzustand* ausdrücken. Für „fault" werden wir dann die Übersetzung *Fehlerursache* verwenden, wenn der Unterschied zu den beiden anderen Bedeutungen betont werden soll; ansonsten vereinfacht *Fehler*.

Wir beschreiben die drei Begriffe und ihre Zusammenhänge zunächst nur informell, wie sie in oben genannter Literatur dargestellt werden und geben eine Klassifizierung an, die die verschiedenen Aspekte der Begriffe beleuchtet. Wir werden dann in Abschnitt 2.1.3 auf der Basis eines einfachen, aber allgemeinen Systemmodells die Fehlerbegriffe konkreter als Abweichungen eines *Ist* von einem *Soll* bezüglich verschiedener Aspekte eines Systems charakterisieren und systematisch darstellen. Eine entsprechende formale Definition dieser Begriffe werden wir schließlich in Kapitel 4.3 vorstellen.

Failure

Ein *(System-) Versagen* (engl. *failure*) liegt vor, wenn das Verhalten eines Systems nicht das Verhalten ist, das in der Spezifikation des Systems gefordert wird. Ein Versagen kann demnach nur während des Betriebes eines Systems auftreten (oder beim Versuch seiner Inbetriebnahme), und auch nur dann festgestellt werden. Die potentielle An- bzw. Abwesenheit von Versagen macht letztendlich die Korrektheit eines Systems aus. Kann man die Abwesenheit von Ausfällen in allen Systemabläufen garantieren, so sind das System und all seine Komponenten vollständig korrekt oder auftretende (lokale) Fehler beliebiger Ausprägung können (auf maskierende Weise) toleriert werden.

Der Begriff des Versagens eignet sich demnach gut als Maß zur Bewertung eines Systems bzgl. seiner Fehleranfälligkeit. Hat man keinerlei Information über die innere Struktur, sieht man das System also in einer *Black-Box-Sicht*, so ist der Begriff des Versagens der einzige, der sinnvoll angewendet werden kann.

In der deutschen Sprache wird *failure* genauer differenziert in *Versagen* und *Ausfall*. Ein Ausfall ist ein Versagen, das mit einer physikalischen und permanenten Verände-

rung einhergeht. Ein durchgebrannter Baustein in einer elektronischen Schaltung ist ein Beispiel für einen Ausfall.

Eine *malfunction* (übersetzbar mit *Fehlfunktion*) verstehen wir als eine inkorrekte Auftragsausführung. Auch bei einer auftretenden Fehlfunktion liegt also eine sichtbare Abweichung vom erwünschten Verhalten vor. Damit können wir eine Fehlfunktion als spezielle Form des Versagens bezeichnen. Während man ein Versagen meist als endgültig und fatal betrachtet, so dass eine Reparatur oder der Ersatz des Systems erforderlich wird, interpretiert man eine Fehlfunktion eher als eine kurzfristige, harmlosere und tolerierbare Abweichung vom spezifizierten Verhalten. Wir verzichten im Rahmen dieser Arbeit auf eine derartige Bewertung der Schwere eines Versagens, und differenzieren daher nicht weiter zwischen *failure* und *malfunction*.

Error

Den Begriff des *Fehlerzustands* (oder auch *Fehlzustandes*, engl. *error*) läßt sich nur bei zustandsbehafteten Systemen definieren. Ein Fehlerzustand wird meist verstanden als ein Zustand des Systems, der vom eigentlich erwünschten Systemzustand abweicht. Ein falscher Wert einer Variable, falsche Daten in einem Datenbankeintrag oder ein falscher Kontrollzustand sind Beispiele für Fehlzustände.

Ein Fehlzustand kann, muß aber nicht zu einem Systemversagen führen. Wird ein falscher Wert an Stelle eines korrekten ausgegeben oder ergibt sich durch einen anderen Kontrollzustand ein nach außen sichtbares verändertes Verhalten, so liegt tatsächlich ein Versagen vor. Wird jedoch ein falscher Wert durch einen korrekten Wert überschrieben, bevor ersterer Wirkung zeigen konnte, kann der vorübergehend vorliegende Fehlerzustand nicht zu einem Versagen führen.

Fault

Als *Fehlerursache* (engl. *fault*) bezeichnet man den Grund, der aus einem System ein fehlerhaftes macht. Die Fehlerursache ist der eigentliche Unterschied zwischen einem fehlerhaften System und seiner korrekten Fassung. Dieser Defekt kann auf allen Ebenen in beliebigen Bestandteilen des Systems liegen: Es können Unzulänglichkeiten in der Hardware oder Software oder/und im Design eines Systems sein, in der Beschreibung der Anforderungen, in der Spezifikation, in der Implementierung, in den Umgebungsannahmen und allen anderen Aspekten eines Systems. Die Klassifikation im nächsten Abschnitt zeigt die vielfältigen Ausprägungen, in denen Fehlerursachen auftreten können.

Die Differenzierung zwischen Fehlerzustand und Fehlerursache wird deutlicher, wenn man den Unterschied bei deren Behebung betrachtet. Ein bekannter und lokalisierter *Fehlerzustand* läßt sich (zur Laufzeit) korrigieren, indem die falschen Werte durch korrekte ersetzt werden. Die *Fehlerursache* ist aber im allgemeinen viel schwieriger zu entfernen, da beispielsweise das Design eines Systems oder sein Programmcode korrigiert werden müssen.

Eine wichtige Klasse von Fehlerursachen sind die *externen Fehler*. Da wir die Umgebung eines Systems als nicht kontrollierbar verstehen, liegt die Fehlerursache hierbei in den

nicht adäquat erfaßten Annahmen an die Umgebung, in der das System eingesetzt wird. Das Verhalten der Umgebung kann in einem solchen Fall zu Fehlerzuständen und zum Versagen des Systems führen. Wird ein solches Versagen durch einen Menschen ausgelöst (der immer zur System*umgebung* gehört), so bezeichnet man diesen externen Fehler als *Fehlhandlung* (engl. *mistake*).

Auch den Begriff der *Störung* wollen wir den Fehlerursachen zuordnen. Nach [20] versteht man unter einer Störung sowohl technisch bedingte Ist-/Soll-Abweichungen von Systemkomponenten als auch Umgebungseinflüsse, die wir bereits als externe Fehler charakterisiert haben. Der Begriff der Störung wird vorwiegend verwendet, um den temporären Charakter einer Fehlerursache zu beschreiben. Dabei unterscheidet er nicht zwischen internen und externen Fehlerursachen.

Zusammenhänge

In der Literatur (beispielsweise in [39, 43, 66]) werden zwei mögliche kausale Zusammenhänge zwischen den drei Fehlerbegriffen angesprochen:

- Eine Fehlerursache bewirkt einen Fehlerzustand.
- Ein Fehlerzustand führt zu einem Versagen.

In beiden Fällen sind die Konsequenzen *möglich*, aber nicht *zwingend*. Eine Fehlerursache kann, muss aber nicht zu einem Fehlerzustand führen. Beispielsweise muß ein „bug“ in einem Programm nicht zu einem falschen Wert führen, wenn der fehlerhafte Programmteil nicht ausgeführt wird, oder in einem konkreten Ablauf keine Folgen zeigt. Auch ein Fehlerzustand muß nicht zu einem sichtbaren Ausfall führen, wenn, wie oben schon erwähnt, ein falscher Wert nie gelesen wird, oder vor dem Lesen durch einen korrekten Wert überschrieben wird. Techniken der Fehlermaskierung und Fehlertoleranz dienen genau dazu, diesen Kausalitätszusammenhang zu durchbrechen.

Die Zusammenhänge der Kausalitäten der drei eingeführten Begriffe sollen an einem Beispiel erläutert werden. Gehen wir aus von einem System, das entsprechend einem Design, also einem Bauplan aus Komponenten zusammengesetzt ist. Im Design sei ein Fehler enthalten, der eine mangelhafte Interaktion zweier Komponenten bewirkt. In einem Ablauf führt dies zu einer inkonsistenten Belegung von Variablen dieser beiden Komponenten. Es liegt also ein Fehlzustand des Systems vor. Es kann sein, dass dieser Fehlzustand nur erkennbar ist, wenn der Gesamtzustand betrachtet wird; aus der Sicht der einzelnen Komponenten ist der Fehlerzustand unter Umständen nicht erkennbar. Im weiteren Ablauf des Systems können die inkonsistenten Werte schließlich zu Ausgaben des Systems führen, die es korrekterweise nicht hätte zeigen dürfen. In diesem Beispiel hat der Designfehler zu einem Fehlzustand geführt, der ein Versagen nach sich zog.

2.1.2 Klassifikationen

Die im vorigen Abschnitt beschriebenen Fehlerbegriffe sind sehr allgemein, und ihre konkreten Ausprägungen in konkreten Systemen können sehr verschiedenartige Eigen-

schaften aufweisen. Wir werden in diesem Abschnitt die Fehlerursachen und Versagen klassifizieren, um so das Spektrum dieser Begriffe aufzuzeigen und eine Strukturierungsmöglichkeit anzubieten.

Eine derartige Klassifizierung kann helfen, einen vorliegenden Fehler und seine Eigenschaften genauer zu beschreiben und bestimmten Fehlerklassen geeignete Umgangsmöglichkeiten zuzuordnen.

Klassifikation von Fehlerursachen

Eine Fehlerursache (im Sinne von *fault*) ist, wie bereits beschrieben, der Unterschied eines Systems von einer als korrekt definierten Fassung. Dieser Unterschied kann verschiedenste Charakteristika aufweisen, von denen wir hier einige angeben wollen. Ist eine Fehlerursache in einem System bekannt, so sollte man in der Lage sein, diese gemäß der folgenden Kriterien einordnen zu können. Die angegebene Klassifizierung faßt die Kriterien zusammen, wie sie sich in der Literatur finden, beispielsweise in [43, 44, 62, 66].

Lokalisierung Ein Fehler kann bezüglich eines Systems *intern* oder *extern* auftreten, falls der verwendete Systembegriff eine klare Grenze zwischen einem Innen und einem Außen eines Systems zuläßt.

Ein Fehler kann *lokal* sein, also einen relativ kleinen, überschaubaren Bereich in beispielsweise einer Spezifikation oder einem Programm betreffen, oder er kann *verteilt* in einem System auftreten, so zum Beispiel in einem Programm als eine Inkonsistenz in der Reihenfolge von Übergabeparametern zwischen der Definition und der Aufrufstelle einer Funktion.

Ein Fehler kann in einer *Spezifikation*, in einem *Systemdesign*, in der *Implementierung* (dem Programmtext) oder auch in den *Umgebungsannahmen* liegen.

Ein Fehler kann in einer einzelnen verwendeten *Komponente* des Systems lokalisiert sein, oder in der *Architektur* des Systems, die fehlerfreie Komponenten auf ungünstige Weise zusammensetzt.

Fehler können in der *Hardware* liegen oder sie können in der *Software* eines Systems enthalten sein. Hardware- und Softwarefehler haben recht unterschiedlich Eigenschaften, die wir in 2.2 ansprechen werden.

Verursacher Für den Verursacher eines Fehlers gibt es zwei Möglichkeiten: Der Fehler kann durch einen *Menschen* verursacht worden sein oder durch die *physikalische Umwelt*. Einflüsse durch elektromagnetische oder radioaktive Strahlungen auf Datenübertragungseinrichtungen, Hitze mit verbundenen Ausdehnungen von Hardwarekomponenten oder Feuchtigkeitseinflüsse auf elektrische Kontakte sind typische Beispiele für physikalische Einflüsse, die auf ein System einwirken können, und sowohl im Betrieb als auch bei der Herstellung eines Systems Fehler bewirken. Fehler durch Menschen können sowohl während der Entstehungszeit durch einen Entwickler in ein System gelangen, als auch während der Laufzeit eines Systems durch einen Benutzer, der durch fehlerhafte Bedienung die Korrektheit eines Systemverhaltens gefährden kann.

Es läßt sich argumentieren, dass eigentlich alle Arten von Fehlern den Menschen als Verursacher haben, denn auch physikalische Fehler sind nur ein Zeichen dafür, dass der Mensch die Technik nicht ausreichend beherrscht und er nicht in der Lage ist, physikalische Fehlerquellen durch entsprechende Gegenmaßnahmen auszuschließen. Aber dennoch hat die Unterscheidung ihre Rechtfertigung: Sie können von einem Entwickler bewußt in Kauf genommen werden, da Maßnahmen zu ihrer vollständigen Vermeidung zu aufwendig wären, wie beispielsweise Abschirmungen von Kommunikationskanälen, oder höhere Qualitätsanforderungen in einem Herstellungsprozeß von Hardwarekomponenten. Stattdessen kann auf Mechanismen zur Tolerierung der physikalischen Fehler ausgewichen werden.

Zeitpunkt des Auftretens Ein Fehler kann zur Zeitpunkt der *Entwicklung* in ein System gelangen, oder während seines Betriebes zur *Laufzeit*. Fehler in der Logik eines Systems, Softwarefehler oder Defizite in einer Systemarchitektur entstehen während der Entwicklung eines Systems. Bedienfehler oder Fehler physikalischer Natur sind dagegen Fehler, die erst zur Laufzeit in einem System auftreten können.

Dauer Bezüglich der Dauer eines Fehler unterscheidet man *permanente* Fehler und *temporäre* Fehler. Ein Fehler heißt permanent, wenn er vom Zeitpunkt seines ersten Auftretens an dauerhaft und ununterbrochen im System verbleibt. Insbesondere sind dauerhafte Fehler, die sogar gleich von Anfang an in einem System enthalten sind, permanent. Temporäre Fehler sind nur zeitweilig in einen System enthalten. Man nennt sie *transient*, wenn sie nur einmal für einen endlichen Zeitraum auftreten, und *intermittierend*, wenn sie wiederholt auftreten, aber dennoch nicht permanent sind.

Grund Man kann zwei Gründe für vorhandene Fehler unterscheiden: Ein Fehler kann *zufällig* in einem System vorhanden sein oder dort *absichtlich* aufgenommen werden. Ein zufälliger Fehler kann sowohl durch Unkenntnis, Fahrlässigkeit als auch Unvermögen des Entwicklers entstehen, der aus diesen Gründen entgegen seiner Bemühungen ein fehlerbehaftetes System schafft. Ein absichtlicher Fehler kann beispielsweise für Testzwecke aufgenommen werden, oder aber auch in krimineller oder böswilliger Absicht, um einen Schaden zu erzeugen oder Daten ausspähen zu können.

Schwere Die Schwere eines Fehlers kann nach verschiedenen Kriterien bewertet werden. Fehler können in Kategorien eingeordnet werden, wie beispielsweise *unkritisch*, *kritisch* und *katastrophal*. Unkritische Fehler betrachtet man als so harmlos, dass sie nur Auswirkungen haben, die keiner intensiven Beachtung bedürfen. Kritische Fehler dagegen beeinträchtigen die Systemleistung so erheblich, dass Gegenmaßnahmen zu ergreifen sind. Die schlimmste Klasse sind katastrophale Fehler. Diese sind absolut untolerierbar und mit allen Mitteln zu vermeiden.

Weitere Möglichkeiten zur Bewertung der Schwere von Fehlern bestehen in der Ermittlung der Kosten, die zur Deckung der durch einen Fehler verursachten Schäden notwendig wären oder werden. Man kann das Maß der Tolerierbarkeit von Fehlern angeben, indem das erreichbare Niveau der Fehlertoleranz (besprochen in

Abschnitt 2.2) bei Vorliegen der Fehler angegeben wird. Fehler können auch nach dem Aufwand bewertet werden, der für ihre Behebung notwendig ist.

Die angegeben Charakteristika kommen nicht in allen Kombinationen vor. Es gibt beispielsweise keine absichtlichen, physikalischen Fehler, da eine Absicht nur einem Menschen zugesprochen werden kann. Es gibt ebensowenig temporäre Softwarefehler, die während der Laufzeit des Systems auftreten. Meist treten die Kriterien in wiederkehrenden Gruppierungen auf. Beispiele sind

- Softwarefehler, die von Menschen begangen wurden, zur Entwicklungszeit auftreten, permanent im System enthalten sind und versehentlich eingefügt wurden, und
- Fehler in der Datenübertragung, die einen physikalischen Ursprung haben, zur Laufzeit und nur temporär auftreten und auch nicht absichtlich verursacht wurden.

Klassifikation von Versagen

Für das Versagen (*failure*) von Systemen oder Komponenten finden sich Klassifizierungen in [23, 27, 43, 62] , die hier zusammengefaßt dargestellt werden. Ein Versagen ist eine von außen beobachtbare Abweichung des Verhaltens von der Erwartung. Wir unterscheiden nach der *Art*, wie das Verhalten abweicht, klassifizieren nach den *Auswirkungen* und der Einheitlichkeit der *Beobachtbarkeit*.

Art Man kann unterscheiden zwischen wert- und zeitbezogenen Versagen von Systemen. Bei wertebezogenen Ausfällen werden Leistungen zwar zum richtigen Zeitpunkt erbracht, aber die Ergebnisse sind nicht korrekt. Ein Beispiel dafür ist ein *value failure*, bei dem ein falscher Wert als Resultat geliefert wird. Zeitbezogenes Versagen liegt vor, wenn zeitliche Bedingungen nicht eingehalten werden. Man kann hier außerdem die beiden Fälle des *early* und *late timing* unterscheiden.

Die Klasse der *omission* oder *response failures* läßt sich nicht eindeutig als wert- oder zeitbezogenes Versagen bezeichnen, da das Ausbleiben eines Ergebnisses sowohl als falscher Wert als auch als ein extrem verspätetes Ergebnis interpretiert werden kann. Ein *Totalversagen* beschreibt den völligen Ausfall eines Systems, das gar keine Ausgaben mehr erzeugt. Ist der Totalausfall erkennbar, kann dies mit *fail-stop failure* bezeichnet werden. Ist der Ausfall nicht unmittelbar erkennbar, sondern sind eigene Mechanismen außerhalb der ausgefallenen Komponente nötig wie beispielsweise Timer, so spricht man von *fail-silent*- oder *crash*-Versagen. Ist nach einem Totalausfall ein Neustart möglich, so kann im Falle eines *pause crash failure* das System in dem Zustand seine Aktivität fortsetzen, in dem das Versagen auftrat (oder einem Zustand kurz vor diesem). Startet das System ganz neu im Grundzustand, also mit völligem Informationsverlust, nennt man dies *amnesia crash failure*. Führen die Auswirkungen eines Versagens zu einer Inkonsistenz von Daten im System, wird ein Fehler *corrupting* genannt. Die (gravierendste)

Klasse der sogenannten *byzantine failures* beschreibt ein chaotisches Versagen, bei dem beliebiges Verhalten möglich wird, also keinerlei Information über das Systemverhalten mehr verfügbar ist.

Auswirkungen Die Auswirkungen von Versagen können (wie Fehler) bewertet werden nach ihrem Schweregrad, und beispielsweise in *harmlos* (oder unkritisch), *kritisch* und *katastrophal* eingestuft werden. In komponierten Systemen ist es interessant, die Auswirkungen des Versagens von Komponenten auf das Gesamtverhalten zu untersuchen. Dazu muss es möglich sein, aus dem Verhalten von Komponenten und der Kenntnis des Systemdesigns das Verhalten des Gesamtsystems ermitteln zu können.

Beobachtung Die Beobachtung eines Ausfalls kann für verschiedene Beobachter *konsistent* oder *inkonsistent* sein. Nicht immer nehmen alle Beobachter einen Ausfall auf die gleiche Weise wahr. So kann ein Versagen für einen Systembenutzer erhebliche Einschränkungen darstellen, während es für einen anderen Benutzer vollkommen unbemerkt bleibt, da nur Funktionalitäten betroffen sind, die von ihm nicht in Anspruch genommen werden. Aber auch bei Verwendung des gleichen Funktionalität können durch unterschiedliche Granularitäten der Zeitwahrnehmung von Benutzern Versagen unterschiedlich wahrgenommen werden. Versagen mit konsistenter Beobachtbarkeit stellen im allgemeinen ein kleineres Problem dar, wenn Versagen im Rahmen einer (automatisierten) Fehlerbehandlung *erkannt* werden sollen.

2.1.3 Fehler als Soll/Ist-Abweichung

Die bisher angegebenen Klassifizierungen stellen das breite Spektrum an Interpretationsmöglichkeiten für den Fehlerbegriff dar, wie sie in der Literatur gefunden werden können. Wir wollen in diesem Abschnitt verschiedene Fehlerbegriffe durch Abweichungen eines *Ist* von einem *Soll* in Bezug auf verschiedene Aspekte eines Systems charakterisieren. Wir beschreiben dazu zunächst ein einfaches Systemmodell und erläutern daran die Attribute eines Systems, die uns im Rahmen dieser Arbeit interessieren. Die bereits beschriebenen Begriffe *Fehlerursache*, *Fehlerzustand* und *Versagen* können wir dann als Abweichungen dieser Attribute beschreiben. In den folgenden Kapiteln werden wir diese Sichtweise präzisieren, indem wir formale Definitionen dieser drei Begriffe erarbeiten.

Wir betrachten Systeme, die aus Komponenten zusammengesetzt sein können. Zur Modellierung von Fehlern wollen wir auf eine Teilkomponente des Systems fokussieren, um darin enthaltene lokale Fehler zu modellieren und ihre Wechselwirkungen sowohl mit dem restlichen System als auch mit der Umgebung explizit darzustellen und zu untersuchen. Dies führt zu einer Sicht auf Systeme, wie sie in Abbildung 2.1 dargestellt ist:

Dargestellt ist darin ein System, das in eine Teilkomponente S und ein Restsystem S' untergliedert ist. Die Systemumgebung wird durch E repräsentiert. Die Teilkomponente S wird im Vergleich zum Restsystem S' hervorgehoben und damit in das Zentrum der

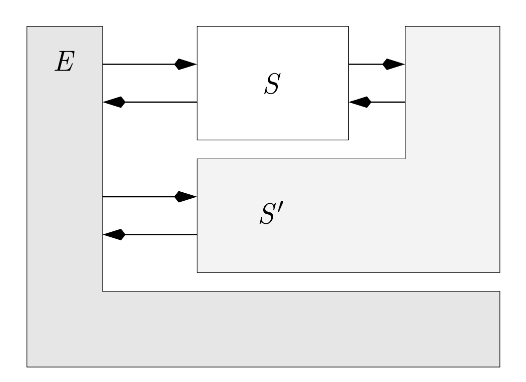


Abbildung 2.1: Zusammengesetztes System mit Umgebung

Betrachtungen gerückt. Die Komponente S interagiert sowohl mit dem Restsystem S' als auch direkt mit der Umgebung, während natürlich auch S' mit E interagiert.

Durch die Wahl der Grenze zwischen S und S' kann jeder beliebige Teil eines Systems fokussiert werden: S kann das Gesamtsystem umfassen, mehrere zusammengefaßte Teilkomponenten oder auch nur eine Einzelkomponente. Im Rahmen einer Fehlermodellierung können damit Fehler eines Systems auf verschiedenen Ebenen – von lokalen bis zu systemglobalen Fehlern – untersucht werden.

Der Unterschied zwischen E und S' besteht im wesentlichen darin, dass E die Systemumgebung beinhaltet, auf die ein Entwickler im allgemeinen keinen Einfluß hat, während S' ein Teil des zu konstruierenden Systems ist, das im Entwicklungprozeß noch entsprechend gestaltet werden kann. Aus der Sicht der Komponente S stellen sowohl S' als auch E externe Systemteile dar, die sich prinzipiell nicht mehr voneinander unterscheiden. Sind wir also an der Fehlermodellierung bezüglich S interessiert, können wir das Systemmodell noch etwas vereinfachen: Wir fassen E und S' zusammen zu einer Umgebung U des Systems S, wie in Abbildung 2.2 dargestellt. Die Interaktion zwischen S' und E stellt in diesem Modell nur noch eine interne Aktivität von U dar[1].

Unser einfaches Systemmodell identifiziert also ein System S, eine Umgebung U und eine Schnittstelle zwischen diesen. Wir können an diesem Modell nun verschiedene Attributklassen charakterisieren.

Durch die Schnittstelle und die klare Abgrenzung von internen und externen Merkmalen können wir zwei Sichten auf ein System unterscheiden: In der *Black-Box Sicht* auf ein System hält man seine Interna vollständig verborgen; nur seine Schnittstelle und Außenwirkung werden betrachtet. In der *Glass-Box Sicht* werden zusätzlich die internen Aspekte eines Systems dargestellt, beispielsweise seine interne Struktur und die Art und Weise, wie es sein Verhalten erbringt. Die Aspekte der Black-Box Sicht sind hierbei mit enthalten, um den Zusammenhang zwischen den Interna und ihrer externen sichtbaren Wirkung herzustellen. Die Black-Box Sicht stellt eine abstraktere Sichtweise dar, da sie im Gegensatz zur Glass-Box Sicht Details verbirgt.

In jeder dieser beiden Sichten lassen sich die Systemmerkmale in *statische* und *dyna-*

[1] Den Operator $\otimes$ zur Komposition werden wir in Abschnitt 3.7 formal definieren.

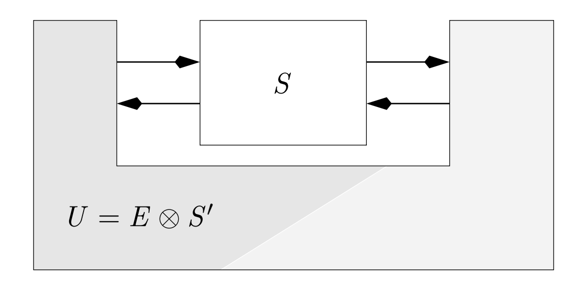


Abbildung 2.2: Vereinfachtes Systemmodell

mische Aspekte untergliedern. Damit erhalten wir die in Abbildung 2.3 tabellarisch dargestellten Attribute eines Systems.

Die Black-Box Sicht umfaßt die Schnittstelle und das (von außen sichtbare) Verhalten des Systems. Die Schnittstelle bleibt in einem Systemablauf unverändert, und ist daher ein statischer Aspekt. Das Verhalten ist ein dynamischer Systemaspekt, wobei wir unterscheiden zwischen dem Verhalten des Systems selbst und den Voraussetzungen an das Verhalten der Umgebung. In vielen Fällen kann ein System seine Leistung nur dann erbringen, wenn die Umgebung gewisse Voraussetzungen erfüllt. Diese werden meist in einer *Umgebungsannahme* formuliert.

Der statische Anteil der Glass-Box Sicht enthält den internen Aufbau eines Systems. Dazu gehören beispielsweise sein Design, sein Code und – soweit vorhanden – auch die Beschreibung interner Teilkomponenten mit ihren Interaktionen. Da wir im Rahmen dieser Arbeit auf die Untersuchung sogenannter dynamischer und mobiler Systeme verzichten, sind all diese Merkmale den statischen zuzuordnen. Die Abläufe eines Systems stellen einen dynamischen Aspekt dar. Ein Ablauf eines Systems besteht aus einer Folge von (Kontroll- und Daten-) Zuständen. Wir ordnen die Zustände und ihre Übergänge also den dynamischen Merkmalen der Glass-Box Sicht zu.

Die beschriebenen Aspekte eines Systems sind nicht unabhängig voneinander, sondern müssen auf vielfältige Weise zusammenpassen: Abstraktionen der Zustandsfolgen aus den Abläufen eines Systems werden als sein Black-Box Verhalten sichtbar, das Umgebungsverhalten hat durch Systemeingaben unmittelbaren Einfluß auf die Abläufe, das Systemdesign legt die möglichen Zustandswechsel fest und das Systemverhalten inklusive seiner Interaktionen mit der Umgebung muss mit der Schnittstelle verträglich sein, um einige Beispiele der Bezüge zu nennen. Es ist die Aufgabe eines Entwicklers, eine derartige Konsistenz innerhalb der verschiedenen Merkmale und Sichten auf ein System zu erreichen.

In vielen Fällen liegen die Bestandteile der Black-Box Sicht als Spezifikation vor. Die internen statischen Bestandteile des Systems werden dann entsprechend konstruiert, und die Abläufe ergeben sich daraus meist aufgrund einer zugrundeliegenden Laufzeitumgebung. Im Idealfall ist am Ende eines Entwicklungsprozesses eine konsistente Definition aller Aspekte erreicht und sogar formal bewiesen. Wir definieren diese Sicht als die *Soll*-Beschreibung eines Systems.

Die verschiedenen Arten von Fehlern können wir nun anhand dieser Darstellung gut

<table>
<tr><th>statisch</th><th colspan="2">dynamisch</th><th></th></tr>
<tr><td>Schnittstelle</td><td>System-verhalten</td><td>Anford. an Umgebungs-verhalten</td><td>Black-Box-Sicht</td></tr>
<tr><td>Systemstruktur, Code, Design</td><td colspan="2">Abläufe als Folge von Systemzuständen</td><td>Glass-Box-Sicht</td></tr>
</table>

Abbildung 2.3: Merkmale eines Systems

illustrieren. Weichen Merkmale eines vorliegenden *Ist*-Systems von einem geforderten *Soll*-System ab, so liegen Fehler vor. Mit Hilfe von Abbildung 2.4 stellen wir die Fehlerbegriffe nun einheitlich im Rahmen unserer Systemsicht vor. Jeder Soll-/Ist-Abweichung der fünf Merkmalsklassen können wir entsprechende Fehlerbegriffe zuordnen.

Unterscheidet sich eine *Ist*-Schnittstelle von einer intendierten Schnittstelle eines *Soll*-Systems, so führt dies zu einer *Inkompatibilität* zwischen dem System und seiner Umgebung. Beispielsweise kann es dann passieren, dass von dem System Nachrichten empfangen werden, die in der Schnittstellenspezifikation nicht enthalten waren und beim Systementwurf daher nicht berücksichtigt wurden. Das System kann daher auf diese Nachrichten nicht reagieren. Im Sinne der Klassifizierung aus Abschnitt 2.1.2 ist eine derartige Fehlerursache verteilt als Inkonsistenz im Gesamtsystem vorhanden, ist durch den Entwickler vermutlich während der Entwicklungszeit durch Fahrlässigkeit verursacht und ist dauerhaft im System vorhanden. Die Schwere dieser Fehlerursache kann nur im Einzelfall beurteilt werden.

Ein *Systemversagen* können wir in Übereinstimmung mit den bereits vorgestellten Begriffen als eine sichtbare Soll-/Ist-Abweichung im Systemverhalten definieren.

Erfüllt die Umgebung nicht die Erwartungen, die an sie in der Umgebungsannahme gestellt wurden, bezeichnen wir dies als einen *externen Fehler*. Es liegt also eine Abweichung zwischen der in der Systembeschreibung enthaltenen Umgebungsannahme und dem konkret beobachteten Umgebungsverhalten vor. Die Abweichung kann sowohl in einer fehlerhaften Erfassung der Umgebungsannahmen als auch im „wirklichen“ Versagen der Umgebung liegen. Wir betrachten Diskrepanzen zwischen einem Ist und einem Soll, ohne dabei unmittelbar Verantwortungen zuzuweisen. Eine derartige Abweichung bezeichnen wir im Einklang mit den Begriffen der Literatur wieder als *Fehlerursache*.

Sehr häufig sind Fehlerursachen in einer Soll-/Ist-Abweichung im System selbst zu finden: Fehler im Design oder im Code eines Systems sind typische Beispiele. Aufgrund ihres statischen Charakters sind diese *Fehlerursachen* permanent und gelangen zur Entwicklungszeit in das System, sind aber unter Umständen nur schwer zu lokalisieren und

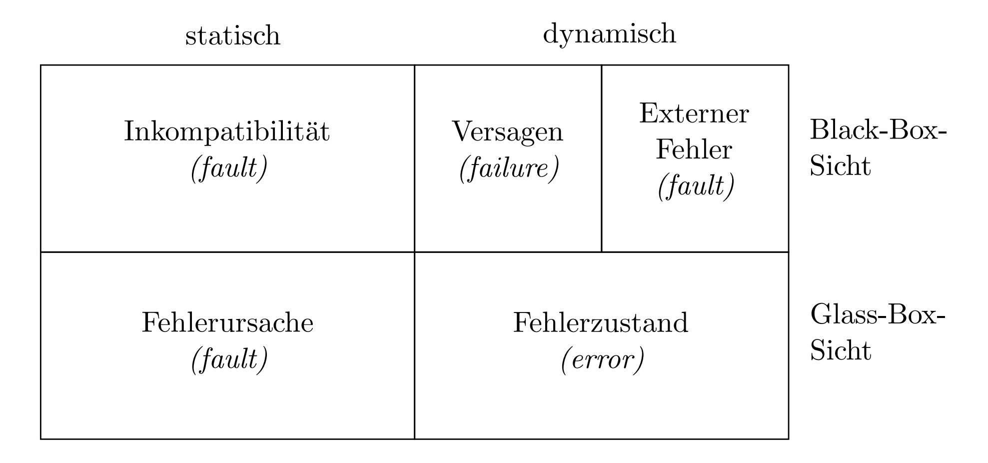


Abbildung 2.4: Fehlerbegriffe für Soll-/Ist-Abweichungen

werden oft erst im Systemablauf in Form von ausgelösten Fehlerzuständen oder Systemversagen erkannt.

Abweichungen eines vorliegenden Ist-Systemzustandes von einem erwarteten Soll-Zustand können wieder in Übereinstimmung mit den etablierten Begriffen als *Fehlerzustand* bezeichnet werden.

Die in Abbildung 2.4 dargestellte Tabelle gibt uns einen Überblick, welchem Fehlerbegriff eine vorliegende Abweichung eines Ist von einem Soll zuzuordnen ist. In Abhängigkeit des Merkmals, das eine vorliegende Diskrepanz aufweist, kann der zugeordnete Fehlerbegriff ermittelt werden. Wir werden alle Aspekte eines Systems in den formalen Definitionen in Kapitel 3 unmittelbar wiederfinden. Die Systemschnittstelle wird in Abschnitt 3.3 definiert, das Verhalten eines Systems in Abschnitt 3.4. Das Umgebungsverhalten wird in Abschnitt 3.5 behandelt. Internes Systemdesign wird in Kapitel 3.6 mit Transitionssystemen dargestellt, die die entsprechenden Systemabläufe als dynamischen Anteil definieren. Eine wichtige Frage dabei ist, wie konkrete Systeme und ihre Fehler zu beschreiben sind. Wir werden dazu geeignete Beschreibungstechniken anbieten. Zur Darstellung der Soll-/Ist-Abweichungen führen wir in Kapitel 4 den Begriff der *Modifikationen* ein, der für alle relevanten Merkmale eines Systems Ausdrucksmöglichkeiten der Fehlerbegriffe bietet.

2.1.4 Umgang mit Fehlern

Nach der Klärung der grundlegenden Begriffe wollen wir nun den Umgang mit Fehlern diskutieren. Es ist heutzutage weitgehend akzeptiert, dass Fehler in komplexen Systemen nicht zu vermeiden sind oder nur mit einem erheblichem Aufwand, der nicht mehr in einer rechtfertigbaren Relation zu potentiellen Schäden steht, die durch tatsächlich auftretende Fehler verursacht werden würden. Fehler sind ein Teil des Verhaltens, der zwar unerwünscht, aber dennoch unvermeidbar ist, und daher in einer Systementwicklung berücksichtigt werden muß.

Im Umgang mit Fehlern können wir zwischen zwei Phasen im Lebenszyklus eines Systems unterscheiden, der *Entwicklungszeit* und der Zeit während des Betriebes eines Systems, seiner *Laufzeit*.

Entwicklungszeit

Der offensichtlich ideale Umgang mit Fehlern ist der Ansatz, jegliche Fehler zu vermeiden, d.h. nur absolut fehlerfreie System zu erstellen und zu betreiben. Inbesondere für Softwaresysteme ist dieser Ansatz lohnenswert, da Software nicht altert, keine Verschleisserscheinungen zeigt und auf nahezu perfekte Weise replizierbar ist. Ist also ein fehlerfreies Software-System einmal geschaffen, bleibt es fehlerfrei.

Die Disziplin des Software- und Requirement-Engineerings erarbeitet Modelle, Vorgehensweisen und Techniken, die dieses Ziel verfolgen. Anforderungen an ein System müssen korrekt erfaßt werden können, präzise Spezifikationen formuliert, ein geeignetes Systemdesign erstellt, Module implementiert und integriert werden. Dies hat unter Einhaltung von Einschränkungen verschiedener Ressourcen wie finanzieller Mittel, Arbeitskraft und Zeit zu geschehen. Ein Hilfsmittel sind formale Methoden, die nicht nur eine präzise Sprache zur Formulierung von Systemen und ihrer Eigenschaften bieten, sondern auch einen Kalkül zur Verifikation von Eigenschaften. Eine Systementwicklung wird typischerweise durch Werkzeuge unterstützt, die zum Beispiel Konsistenzen und Vollständigkeiten zwischen verschiedenen Dokumenten überprüfen, Code aus Spezifikationen und Prototypen generieren und Gruppenarbeit unterstützen.

Trotz gewisser Fortschritte dieser Disziplin in den letzten Jahren kann aufgrund der hohen und weiter wachsenden Komplexität der Systeme selten vollkommene Fehlerfreiheit erreicht werden. Ein Entwicklungsprozeß umfaßt in der Regel daher auch *Reviews* von Beschreibungs- und Spezifikationsdokumenten, mit denen versucht wird, Fehler in einem System, seinen Anforderungen und seiner Implementierung aufzudecken, um diese anschließend zu beheben. Systeme und ihre Komponenten werden, bevor sie in ihrer eigentlichen Umgebung zum Einsatz kommen, auch immer durch verschiedenartige *Tests* geprüft, um somit nach enthaltenen Fehlern zu suchen. Versagt ein Testlauf, werden die verantwortlichen Fehler lokalisiert und entfernt.

Aber auch im laufenden Betrieb können noch Fehler festgestellt werden. Typischerweise wird ein System dann durch die Entwickler durch Erstellung einer neuen Version oder das Ausliefern von *Patches* nachgebessert. Wir ordnen diese Tätigkeit auch noch der Entwicklungszeit zu.

Laufzeit

Die genannten Tätigkeiten während der Entwicklungszeit zielen alle auf ein fehler*freies* System ab. Beim Umgang mit Fehlern während der Laufzeit eines Systems akzeptiert man aber das Eingeständnis, dass Fehler in einem System enthalten sein können – seien es Softwarefehler, die sich durch Defizite in der Entwicklung ergeben haben, Hardwarefehler, die durch Probleme in der Herstellung, durch Verschleiß bzw. Alterung oder andere physikalische Einflüsse verursacht werden, oder andere unberücksichtigte äußere

Einflüsse wie beispielsweise Umgebungsfehler oder Viren.

Ein Umgang mit derartigen Fehler zur Laufzeit muss durch das System selbst erfolgen. Wir können diesbezüglich folgende Systemaktivitäten unterscheiden:

- *Fehlerentdeckung.* Das System soll in der Lage sein, einen Fehler (im Sinne von Fehlzustand oder Versagen) zu entdecken, um darauf reagieren zu können. Eine Fehlerursache hingegen kann ein System normalerweise nicht selbst feststellen. Wären dafür Mechanismen vorhanden, hätte man sie bereits vor Inbetriebnahme des Systems verwenden können, um den Fehler zu beheben. Das System soll während seiner Laufzeit aber die *Wirkung* einer Fehlerursache entdecken können. Dazu werden wir einige Techniken in Abschnitt 2.2.2 vorstellen.

- *Fehlerlokalisierung.* Zwischen dem Auftreten eines Fehlers und seiner Entdeckung können sich die Folgen des Fehlers bereits über das System ausgebreitet haben. Techniken der Fehlerlokalisierung sollten in der Lage sein, diejenigen Systemteile identifizieren zu können, die betroffen sind oder betroffen sein könnten. Nur so ist es möglich, die Auswirkungen zu beheben und eine Ausbreitung von vornherein zu verhindern oder zumindest einzuschränken. Die Fehlerlokalisierung kann im Idealfall auch ermitteln, welche Fehlerursache vorliegt.

- *Fehlermeldung.* Wird ein Fehler entdeckt, so ist die Meldung an andere Systemteile oder an die Umgebung des Systems (zum Beispiel seinen Benutzer) die minimale Reaktion, die vom System auf alle Fälle geleistet werden muss. Eine Fehlermeldung kann mit einer Deaktivierung des Systems (oder eines Teiles) verbunden sein, wenn andere Reaktionen zur Fehlerbehebung nicht möglich sind.

- *Fehlerbehebung.* Im Idealfall kann ein System einen Fehler(zustand) und seine Auswirkungen nach seiner Erkennung korrigieren. Dies kann mit einer temporären Veränderung des Systemverhaltens verbunden sein, im optimalen Fall aber sogar nach außen nicht sichtbar werden. Wir werden auf einige technische Möglichkeiten in Abschnitt 2.2.2 eingehen.

Mit den genannten Mechanismen wird es möglich, ein *fehlertolerantes* System zu entwickeln, das trotz enthaltener und auftretender Fehler ein akzeptables Verhalten zeigt. Wir widmen diesem Thema den folgenden Abschnitt.

Einen Sonderfall im Umgang mit Fehlern stellen die sogenannten *selbststabilisierenden Systeme* [61] dar. Diese zustandsbasierten Systeme streben von (nahezu) jedem beliebigen Zustand immer zu Zuständen, die als korrekt definiert sind, und kommen unter Umständen ohne explizite Fehlerentdeckung und -lokalisierung aus. Die Fehlerbehebung ist damit ein integraler Bestandteil der Funktionsweise eines derartigen Systems. Diese Systeme basieren auf speziellen Eigenschaften des zugrundeliegenden Algorithmus und sind in der Praxis (nach [28]) nur sehr selten anzutreffen.

Die Techniken, die die Fehlertoleranz eines Systems ermöglichen, werden natürlich auch schon während der Entwicklungszeit integriert, so dass in gewisser Weise der Umgang mit Fehlern nicht nur während der Laufzeit erfolgt, sondern vorbereitend bereits zur

Entwicklungszeit. Dennoch besteht hier ein qualitativer Unterschied: Nur der automatisierte Umgang mit Fehlern zur Laufzeit kann (eventuell unvermeidbaren) Fehlern begegnen, die erst zur Laufzeit auftreten oder während der Entwicklungszeit übersehen wurden. Die entsprechenden Techniken unterscheiden sich erheblich von Entwicklungstechniken zur Vermeidung von Fehlern.

2.2 Fehlertoleranz

Als Fehlertoleranz wird (zum Beispiel in [20]) die Eigenschaft eines System bezeichnet, trotz (intern oder extern) auftretender Fehler noch immer ein gewünschtes oder zumindest davon nur geringfügig abweichendes Verhalten aufzuweisen. Eine Aussage über die Fehlertoleranz eines Systems muss also zwei Teilaussagen umfassen:

- *Die Menge der tolerierten Fehler muss angegeben werden.* Im allgemeinen können nicht beliebige Fehler (zum Beispiel Fehler mit extrem fataler Wirkung) toleriert werden, sondern nur eine zu definierende Klasse von Fehlern, von denen man annimmt, dass das System ihnen ausgesetzt sein kann. Diese Menge von Fehlern wird häufig auch als *Fehlermodell* bezeichnet. Über Fehler, die *nicht* in dieser Klasse sind, wird keine Fehlertoleranz-Aussage gemacht. So wird für manche Fehler beispielsweise angenommen, dass ihr Auftreten zu unwahrscheinlich ist, als dass sie berücksichtigt werden müssen.

- *Die zulässige Abweichung vom fehlerfreien Verhalten ist zu definieren.* Nur im idealen Fall bleiben die Auswirkungen von Fehlern nach außen unsichtbar. Im allgemeinen gibt es eine Veränderung, beispielsweise in der Performanz eines Systems, in seinem zeitlichen Verhalten oder auch in einer (tolerierbaren) Abweichung von berechneten Werten vom eigentlich korrekten Resultat. Diese in der Literatur oft als *graceful degradation* (sanfte Leistungsminderung) bezeichnete Abweichung ist für eine klare Aussage zur Fehlertoleranz explizit zu machen.

Für ein System können auch mehrere, gestaffelte Aussagen zur Fehlertoleranz gültig sein: Für verschiedene Mengen von Fehlern weicht das Verhalten auf unterschiedliche Weise ab. So können beispielsweise für harmlose Fehler keine Auswirkungen sichtbar werden, während für gravierendere Fehler nur noch sicherheitskritische Eigenschaften erfüllt bleiben.

Die stärkste Form von Fehlertoleranz ist die sogenannte *maskierende Fehlertoleranz*. Sie stellt sicher, dass bezüglich einer definierten Menge von Fehlern das Verhalten unverändert bleibt, die Fehler also keine Wirkung zeigen können.

Die in der Klassifizierung von Versagen aufgeführten Beispiele können für explizite Aussagen über Fehlertoleranz verwendet werden. So können Komponenten eines Systems bestimmte Klassen von Versagen als Fehlermodelle zugeordnet werden, und damit eine Aussage über die Menge der zu tolerierenden Fehler gemacht werden. Andererseits kann die Abweichung vom erwarteten Verhalten des Gesamtsystems durch diese Klassen beschrieben werden.

Beispiel 2.2 Sei ein einfaches System gegeben, das der Nachrichtenübermittlung dient, bestehend aus einem Sender, einem Empfänger und zwei Übertragungskanälen. Sender und Empfänger übermitteln Daten mit Hilfe der beiden Kanäle entsprechend dem Sliding-Window-Protokoll (beschrieben beispielsweise in [67]).

Das Protokoll ist in der Lage, den sporadischen Verlust von Nachrichten auf den Kanälen zu tolerieren, so dass nur die Geschwindigkeit der Übertragung beeinträchtigt wird. Spielt das zeitliche Verhalten keine Rolle, kann man also die folgende Fehlertoleranz-Aussage treffen: Das System ist bezüglich *omission failures* der Übertragungskanäle maskierend fehlertolerant. □

Im allgemeinen möchte man aber sehr präzise Aussagen über die Auswirkungen von Fehlern auf ein System machen. Dazu ist es notwendig, sowohl die Systeme, ihre Fehler als auch die Abweichungen vom Normalverhalten unter Einfluß der Fehler zu beschreiben. Mit formalen Techniken kann dabei die höchste Präzision erreicht werden. Im nächsten Abschnitt geben wir einen kurzen Überblick über existierende formale Ansätze, die den Begriff der Fehlertoleranz definieren. Der in den folgenden Kapiteln 3 und 4 vorgestellte Formalismus wird diese Ansätze verallgemeinern und ergänzen.

2.2.1 Formale Ansätze

Wir werden in diesem Abschnitt formale Ansätze und Definitionen zur Fehlertoleranz vorstellen. Wir diskutieren dazu die Ansätze von Zhiming Liu und Mathai Joseph [47], Anish Arora und Sandeep Kulkarni [5] sowie Thomsz Janowski [36], die auch von Felix Gärtner in [28] als die wesentlichen, bekannten Arbeiten identifiziert werden.

Um Fehler und Fehlertoleranz formal ausdrücken zu können, muß zunächst definiert werden, was ein System und sein Verhalten ist. Die genannten formalen Ansätze, die wir im folgenden kurz vorstellen werden, verwenden alle eine ähnliche Fehlermodellierung auf Grundlage verwandter Systemmodelle, die auf Zuständen und Transitionen basieren, und die nun kurz charakterisieren.

Ein *System* befindet sich zu jedem Zeitpunkt in einem Zustand. Ein Zustand besteht im wesentlich aus einer Belegung aller Variablen des Systems mit konkreten Werten. Der Übergang zwischen zwei Folgezuständen wird durch eine Transition beschrieben, die als bewachte Zuweisung ausgedrückt werden kann. Ein Ablauf eines Systems ist eine endliche oder unendliche Folge von Zuständen, in der zwei aufeinanderfolgende Zustände einer legalen Transition des Systems entsprechen. Wir stellen in Kapitel 3.6 ein Modell für Transitionssysteme vor, und verzichten hier daher auf eine detailliertere Darstellung.

Fehler werden in oben genannten Modellen als explizit angegebene Mengen zusätzlicher Transitionen dargestellt, wie bereits 1985 in [22] vorgeschlagen. Auf eine Aussage über die Veränderung der Systemeigenschaften verzichten einige formale Ansätze, da sie Fehlertoleranz nur als maskierende verstehen.

Liu/Joseph Liu und Joseph stellen in ihren Arbeiten [46, 47, 48] Fehlertoleranz nur in der speziellen Form der maskierenden Fehlertoleranz dar. Sie bezeichnen mit P das Nor-

malsystem, und mit $F(P)$ das durch eine definierte Fehlermenge beeinflußte System[2]. Das entstehende System wird fehlertolerant gegenüber einer Spezifikation S genannt, wenn das System alle Eigenschaften von S erfüllt, also gilt

$$F(P) \Rightarrow S$$

Sie lassen im Falle einer nicht gültigen Fehlertoleranz zu, dass ein System durch Integration weiterer Mechanismen fehlertolerant gemacht wird. Die Integration wird durch einen Operator R ausgedrückt, und das verbesserte, weiterentwickelte System durch $R(P)$. Die Fehlertoleranzaussage lautet dann $F(R(P)) \Rightarrow S$.

Janowski Auch Janowski hat in seinen Arbeiten [35, 36, 37, 38] Transitionssysteme als Systemmodell gewählt. Er konzentriert sich ebenfalls auf maskierende Fehlertoleranz und drückt sie mit Hilfe einer Bisimulation aus, die die zusätzlichen Fehlertransitionen berücksichtigt. Ein Prozeß Q ist eine fehlertolerante Implementierung eines (fehlerfreien) Spezifikationsprozesses P, wenn

- jeder Schritt von P durch einen *fehlerfreien* Schritt von Q beobachtungsäquivalent simuliert werden kann, und

- jeder (möglicherweise durch Fehler beeinflußte) Schritt von Q durch einen (natürlich fehlerfreien) Schritt von P simuliert werden kann.

Wichtig ist hierbei, dass ein Schritt von P durch Q simuliert werden kann, ohne dass dabei Fehlertransitionen verwendet werden können. Janowski vertritt die These, dass man sich auf das Auftreten von Fehlern nicht verlassen kann. Fehler passieren seiner Ansicht nach also nichtdeterministisch, unkontrollierbar und unvorhersehbar. Er kann demzufolge mit seinem Ansatz permanente Fehler wie beispielsweise Ausfälle von Komponenten und Designfehler in der Software nicht untersuchen.

Janowski bietet eine erweiterte Version des Prozesskalküls CCS, mit dem es ermöglicht wird, fehlerbehaftete Prozesse zu spezifizieren und Eigenschaften abzuleiten. Er definiert eine Ordnung über Fehlermodelle, die die Schädlichkeit von Fehlern vergleichen läßt. In seinem Modell gilt Monotonie in folgendem Sinne: Toleriert ein System ein gewisse Menge von Fehlern, so toleriert es auch jede Teilmenge davon. Diese Eigenschaft steht in unmittelbarem Zusammenhang mit dem unvorhersehbaren Auftreten von Fehlern. Janowski diskutiert erwünschte Eigenschaften seines Ansatzes, der eine schrittweise Entwicklung fehlertoleranter Systeme ermöglicht. Dazu gehören eine Unabhängigkeit der Entwicklung von Systemkomponenten und die schrittweise Integration von Mechanismen gegen wachsende Fehlerklassen ohne dabei bereits integrierte Techniken unwirksam oder sogar inkorrekt zu machen.

[2]Wir wählen hier nicht die Notation aus dem Originalpapier, sondern eine vereinfachte Version, die die enthaltene Idee aber angemessen ausdrückt. Die Autoren wählen in ihren diversen Arbeiten verschiedene Notationen.

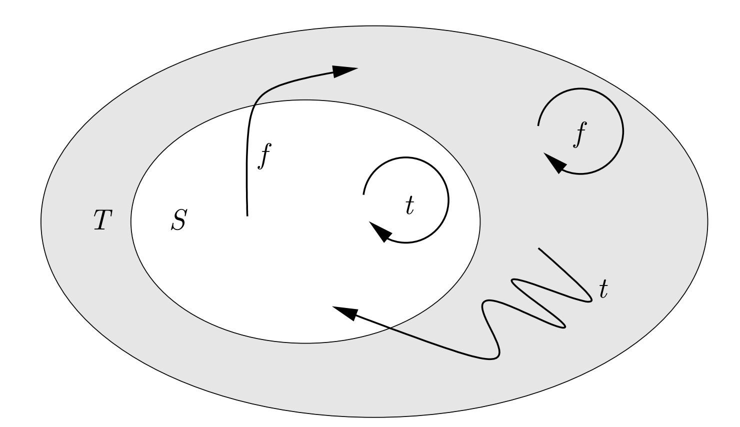


Abbildung 2.5: Normal- und Fehlerzustände

Arora/Kulkarni Im Ansatz von Arora und Kulkarni [2, 3, 41, 4, 5, 42] wird Fehlertoleranz allgemeiner ausgedrückt, da nicht nur maskierende Fehlertoleranz betrachtet wird, sondern auch Abschwächungen von verhaltensbeschreibenden Aussagen.

Ein Zustandsprädikat S ist abgeschlossen für eine Transitionsmenge, wenn alle Transitionen von einem Zustand $s \in S$ wieder in einen Zustand in S führen. Ein Zustandsprädikat beschreibt also eine Invariante. Betrachtet man nun zusätzliche, als Fehler bewertete Transitionen, so können neue Zustände erreichbar werden, die durch ein Zustandsprädikat T charakterisiert werden. Mit diesem Prädikat kann eine Aussage über Fehlertoleranz gemacht werden, denn es beschreibt die Abweichung des Systemverhaltens unter dem Einfluß von Fehlern. Die Autoren formulieren folgende drei Bedingungen, die in Abbildung 2.5 veranschaulicht werden:

- T umfaßt die Normalzustände S, das heißt, es gilt in jedem Zustand, in dem auch S gilt, also $S \Rightarrow T$.

- T ist abgeschlossen unter Normal- und Fehlertransitionen. Das Prädikat T umfaßt also alle Zustände, die bei auftretenden Fehlern erreicht werden können. Das Prädikat T kann als die durch die Fehler induzierte Erweiterung von S verstanden werden, und wird auch als *Fehlerspanne* bezeichnet.

- Die Normaltransitionen t führen schließlich wieder zurück zu Zuständen, für die S gilt. Werden Fehlertransitionen f nicht mehr ausgeführt, stabilisiert sich das System also wieder und kehrt zurück zum erwarteten Verhalten.

Sind diese Bedingungen erfüllt, wird ein System fehlertolerant genannt bezüglich der definierten Fehlermenge und der Fehlerspanne T.

Zwei Spezialfälle dieser Bedingungen führen direkt zu einer einfachen Klassifizierung von Fehlertoleranz: Gilt $S = F$, so wird das Systemverhalten nicht durch die Fehler beeinflußt, es liegt also *maskierende*, ansonsten *nicht-maskierende* Fehlertoleranz vor. Kann ein System von beliebigen Zuständen aus wieder in Normalzustände zurückfinden,

nennt man dies *global stabilisierendes* System, ansonsten nur *lokal stabilisierend* [3]. Ein global stabilisierendes System ist ein *selbst-stabilisierendes* System (vergleiche Abschnitt 2.1.4).

In ihren Arbeiten nutzen die Autoren ihren Ansatz, um sogenannte *Detektoren* und *Korrektoren* als notwendige Bestandteile fehlertoleranter System charakterisieren zu können. Während Detektoren das Auftreten von Fehlern aufdecken, führen Korrektoren nach der Fehlererkennung das System wieder in einen fehlerfreien Zustand zurück. Detektoren sind die minimale Voraussetzung für Fehlertoleranz, mit der *fail-stop* Verhalten erreicht werden kann. Dabei wird beim Auftreten von Fehlern ein System angehalten. Mit Korrektoren können stabilisierende Systeme konstruiert werden. Sind beide Bestandteile für ein Fehlermodell verfügbar, kann maskierende Fehlertoleranz erreicht werden. In [42] beschäftigen sich die Autoren mit dem automatischen Ergänzen von Mechanismen der Fehlertoleranz für fail-stop, maskierender und nicht-maskierender Fehlertoleranz mit Untersuchungen der Komplexität dieser Algorithmen.

Bewertung der Ansätze

Alle Autoren gehen von Systemspezifikationen aus, die als *korrekt* definiert werden. In keinem der Ansätze ist es also möglich, Fehler in den *Anforderungen* an ein System zu formulieren. Dies scheint aber unvermeidbar, da man für eine formale Untersuchung mit der damit erwarteten Präzision eine fixierte Grundlage haben muss, so wie jeder Kalkül von akzeptierten Axiomen und Schlussregeln ausgehen muss.

Da alle genannten Ansätze Fehler durch Transitionen beschreiben, die dem System *hinzugefügt* werden, bleibt das ursprüngliche, fehlerfreie Verhalten immer als eine Verhaltensvariante möglich. Die Fehlertransitionen erweitern die Möglichkeiten durch Ergänzung nichtdeterministisch auswählbarer Alternativen, aber sie schränken das Verhalten nie ein. Diese Ansätze eignen sich also eher zur Modellierung temporärer, physikalischer Fehler, die während der Laufzeit auftreten, und weniger für permanente Softwarefehler, bei denen beispielsweise eine Rechnung in jedem Ablauf falsch durchgeführt wird.

Die vorgestellten Modelle sind geschlossene Systeme und unterscheiden nicht zwischen einem Innen und einem Außen eines Systems. Demzufolge ist es auch nicht möglich, zwischen internen und externen Fehlern eines Systems zu unterscheiden entsprechend der *Lokalisierung* von Fehlern aus Abschnitt 2.1.2. Ferner werden die Zusammenhänge von Fehlern in Komponenten und ihre Auswirkungen auf das Gesamtsystem nicht oder nur oberflächlich untersucht.

Alle formalen Modelle dienen vorwiegend einer grundlegenden Modellierung von Fehlern und Fehlertoleranz in zustandsbasierten Systemen. Ihre Tauglichkeit wird, wie in formalen Methoden häufig, nur durch kleine, eher akademische Beispiele veranschaulicht, so dass Schwierigkeiten in einer konkreten Anwendung nicht unterschätzt werden dürfen. Es ist daher zu vermuten, dass die Techniken in der dargebotenen Form in realen Anwendungen noch nicht brauchbar eingesetzt werden können.

---

[3] Gärtner greift in [28] diesen Ansatz auf und differenziert weiter nach Sicherheits- und Lebendigkeitseigenschaften der Systeme.

Wir werden in Kapitel 4 versuchen, diesen Defiziten zu begegnen. Wir werden dazu auch das Entfernen von Transitionen zulassen, und Fehler auch auf der Ebene der Systemeigenschaften beschreiben. Das Systemmodell wird eine Unterscheidung von internen und externen Fehlern zulassen, und die Zusammenhänge zwischen Komponenten- und Systemfehlern werden untersucht.

2.2.2 Techniken der Fehlertoleranz

Formale Modelle, wie wir sie im vorherigen Abschnitt vorgestellt haben, bieten eine präzise Sprache, mit der Fehlertoleranz und ihre Eigenschaften formuliert und untersucht werden können. Mit dieser Sprache können Techniken, die sich in fehlertoleranten Systemen finden, ausgedrückt und untersucht werden. In diesem Abschnitt wollen wir die Prinzipien fehlertoleranter Techniken beschreiben. Dazu werden wir Möglichkeiten zur Fehlererkennung, Prinzipien der Behebung und eine Klassifikation von Redundanz skizzieren und schließlich jeweils zwei etablierte Techniken für Hardware- und Softwarefehler darstellen.

Eine Voraussetzung für Fehlertoleranz ist die *Fehlererkennung*. Fehler können mit Hilfe von *Replikation* erkannt werden, wenn eine Komponente identisch oder variiert repliziert wird und die Ergebnisse der verschiedenen Versionen verglichen werden. Sind die Ergebnisse nicht identisch oder ist ihre Abweichung außerhalb tolerierbarer Werte, so ist dies ein Indiz für ein aufgetretenes Versagen. Replikation ist zwar mächtig, kann aber erhebliche Ressourcen erfordern und daher zu kostenintensiv sein. Versagen können auch durch *Umkehrungstests* aufgedeckt werden. Von der Ausgabe eines Systems wird auf die Eingaben zurückgeschlossen und die Plausibilität überprüft. Eine *Probe* ist hierfür ein gängiges Beispiel. Sie ist geeignet, wenn die Umkehrung sehr viel leichter zu berechnen ist als die zu berechnende Funktion. So läßt sich beispielsweise die Berechnung des Quadrats viel effizienter durchführen als die Berechnung der Quadratwurzeln. *Konsistenztests* sind eine weitere Technik zur Fehlererkennung. So kann die Konsistenz von Datenstrukturen überprüft werden oder es werden Quittungen bei Datenübertragungen eingefordert. Beispielsweise könnten Sensorenfehler durch zu abrupte Änderungen der gelieferten Daten erkannt werden. Weitere Möglichkeiten zur Fehlererkennung sind zeitliche Überprüfungen, die mit Timern realisiert werden oder Codierungstests, die mit Hilfe von Prüfsummen korrupte Datensätze erkennen.

Nach der Erkennung eines Fehlers folgt idealerweise seine Behebung. Folgende drei Prinzipien können dabei zur Anwendung kommen:

Rückwärts-Fehlerbehebung Bei einem auftretenden Fehler wird zu einem rechtzeitig gespeicherten, korrekten Rücksetzpunkt zurückgesprungen. Die Speicherung muss also regelmäßig erfolgen, so dass ausreichend früh und möglichst zeitnah vor dem möglichen Auftreten von Fehlern ein Rücksetzpunkt gespeichert wird. Der gespeicherte Zustand muss selbst vor Fehlern geschützt sein. Ein Vorteil der Rückwärts-Fehlerbehebung ist die allgemeine Anwendbarkeit, da diese Technik weitgehend systematisch auf viele Systeme angewendet werden kann, unabhängig von ihrer spezifischen Funktionalität. Nachteile sind der relativ hohe Aufwand und Verluste in der Performanz der Systeme. Während bei selten gesicherten

Rücksetzpunkten unter Umständen weit zurückgesetzt und damit lange Abläufe erneut durchgeführt werden müssen, führt häufiges Sichern von Rücksetzpunkten zu einer nicht unerheblichen dauerhaften Verlangsamung des Systems – auch ohne auftretende Fehler. Nicht geeignet ist diese Technik jedoch bei permanenten Fehlern, bei denen ein erneutes Durchführen keine Verbesserung erbringt, und bei Systemen, die viel Interaktion mit ihrer Umgebung aufweisen, die sich nicht oder nur schwer zurücksetzen läßt.

Vorwärts-Fehlerbehebung Tritt ein Fehler auf, wird mit dieser Technik versucht, einen bereits entstandenen Schaden zu ermitteln, und diesen zu beheben, indem das System in einen Zustand versetzt wird, den es ohne die aufgetretenen Fehler hätte erreichen müssen. Möglicherweise kann nur eine abgeschwächte Form der Fehlerbehebung erreicht werden, bei der nur ein Zielzustand angenommen wird, der vom Ideal auf noch akzeptable Weise abweicht. So können beispielsweise bei einem überlasteten System zur Übertragung von Daten Nachrichtenpakete verworfen werden. Diese Technik ist deutlich effizienter als die rückwärts gerichtete, aber nur in Abhängigkeit von der konkreten Anwendung und sehr spezifisch einsetzbar.

Fehlerkompensation Bei der Fehlerkompensation werden Auswirkungen von Fehlern mit Hilfe ausreichender Redundanz sofort vermieden. Beispiele sind fehlerkorrigierende Codes zur Datenübertragung und auch RAID-5 Systeme. Diese Systeme halten ausreichend zusätzliche Information verfügbar, um Fehler nach Lokalisierung eines entstandenen Schadens sofort zu beheben. Die Fehlerkompensation läßt sich als Mischform der Vorwärts- und Rückwärtsfehlerbehebung interpretieren.

Alle Techniken zur Fehlertoleranz benötigen *Redundanz* in einem System. Redundanz wird etabliert durch die Bestandteile eines Systems, die bei Fehlerfreiheit des Systems gar nicht notwendig wären. Redundanz kann in vielen Formen auftreten: Bei *struktureller Redundanz* werden Komponenten des Systems vervielfältigt. *Funktionale Redundanz* impliziert die Integration zusätzlicher Komponenten, die gezielt für die Fehlertoleranz in das System aufgenommen werden. Diese Form kommt in allen fehlertoleranten Systemen vor, da immer Systemteile zur Fehlererkennung, zum Rücksetzen oder Vorausrechnen, zur Fehlerkompensation oder zum Melden von Fehlern notwendig sind. *Informationsredundanz* beinhaltet die Speicherung redundanter Daten im System, beispielsweise als Prüfsummen. *Zeitliche Redundanz* liegt vor, wenn zusätzliche Zeit in einem Systemablauf aufgewendet wird, beispielsweise durch wiederholte Abläufe. Man spricht von *statischer Redundanz*, wenn die redundanten Bestandteile im fehlerfreien Normalbetrieb bereits einen Beitrag zur Systemleistung erbringen. Tun sie dies nur im Ausnahmefall, nennt man dies *dynamische Redundanz*.

Die Wahl einer Technik zum Erreichen von Fehlertoleranz hängt von den Charakteristika der zu tolerierenden Fehler ab. Es besteht hier ein wesentlicher Unterschied zwischen der sogenannten *Softwarefehler-Toleranz* und der *Hardwarefehler-Toleranz*[4]: Die (bau-

[4] Die englischen Begriffe *software fault tolerance* und *hardware fault tolerance* lassen sich auch interpretieren durch Software-Fehlertoleranz bzw. Hardware-Fehlertoleranz. Diese bezeichnen eine Fehlertoleranz, die mit Software bzw. Hardware *erreicht* wird. Wir meinen hier aber die Toleranz von Software-

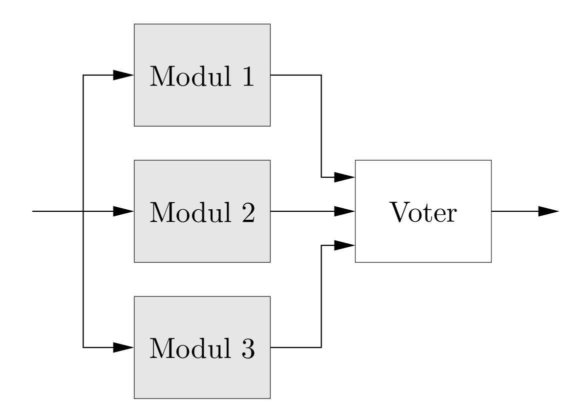


Abbildung 2.6: Triple Modular Redundancy

gleiche) Replikation von Komponenten ist als Maßnahme gegenüber Hardwarefehlern sinnvoll, aber nicht gegenüber Softwarefehlern, die immer Designfehler sind.[5]

Hardwarefehler sind in den meisten Fällen physikalische Fehler, die temporär und eher sporadisch auftreten. Das Auftreten dieser Fehler in verschiedenen, aber baugleichen Komponenten kann statistisch unabhängig sein, wenn sie nur von physikalischen Zufälligkeiten abhängen, und nicht etwa beispielsweise durch die gleichen Umgebungseinflüsse beeinträchtigt werden. Die Auftreten von Fehlern in der Hardware wird durch Alterungs- und Verschleisserscheinungen beeinflußt. Spielen diese Effekte eine Rolle, ist es sinnvoll, eine identische Ersatzkomponente bereitzustellen, die bei Ausfall der eigentlichen Komponente deren Aufgabe übernimmt. Auch permanenten Fehlern in der Hardware kann durch Replikation begegnet werden, wenn diese ihren Ursprung in Defiziten im Herstellungsprozeß haben. Einen Fehler, den eine Komponente hat, wird mit einer meist hohen, oft bekannten Wahrscheinlichkeit eine andere Komponente nicht haben. Bekannte Techniken zur Hardwarefehler-Toleranz mit Replikation sind *Triple Modular Redundancy* und *Standby Spares*, die wir im folgenden kurz skizzieren.

Triple Modular Redundancy Gegeben sei eine Komponente mit einer beliebigen Ein-/Ausgabefunktionalität, bei der man von potentiellem Versagen ausgehen muss. Um eine höhere Robustheit zu erreichen, wird diese Komponente verdreifacht, und alle drei Module mit den gleichen Eingabedaten versorgt. Eine neue Komponente, der *Voter*, liest die Ausgaben der drei Module, und ermittelt durch Mehrheitsbildung ein Ergebnis, das an die Umgebung ausgegeben wird (Abbildung 2.6). Das Versagen von nur einer Komponente kann nach außen maskiert werden. Versagen mehrere Komponenten, kann der Fehler noch erkannt werden. Versagen allerdings mehrere Module auf die gleiche Weise, wird die entstehende Ergebnismehrheit nach außen weitergegeben. Ein präzise Aussage über Fehlertoleranz entsprechend Abschnitt 2.2 muss all diese Fälle berücksichtigen, also für verschiedene Fälle die jeweiligen Auswirkungen beschreiben.

bzw. Hardwarefehlern.

[5]Designfehler in der Hardware spielen hier eine Sonderrolle, und sind aufgrund ihrer Charakteristik eher den Softwarefehlern zuzuordnen.

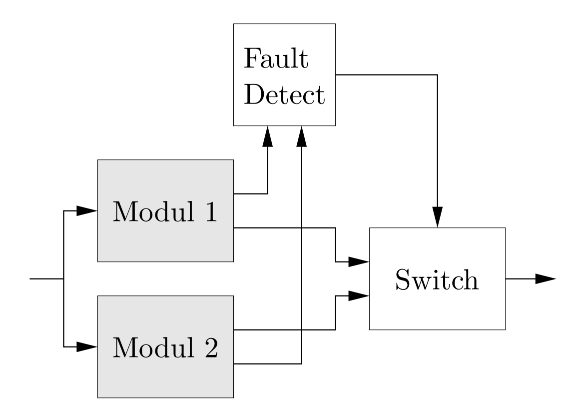


Abbildung 2.7: Reservekomponenten

Es gibt Verallgemeinerungen dieser Technik, zum Beispiel durch die Verwendungen von mehr als drei Modulen oder auch eine Replizierung des *Voters* [66].

Standby Spares Eine weitere, häufig verwendete Technik ist die Verwendung von Reservekomponenten, wie in Abbildung 2.7 dargestellt. Im Normalbetrieb erbringt ein Modul die geforderte Funktionalität. Versagt dieses Modul, wird dies durch eine Fehlerkennungskomponente an den Umschalter signalisiert, der dann nicht mehr die Ausgaben des defekten ersten Moduls, sondern des zweiten Moduls an die Umgebung weiterleitet. Varianten und Verallgemeinerungen finden sich wieder in [66].

Im Gegensatz zu Hardwarefehlern sind Softwarefehler in jedem Fall permanente Fehler, die im Design eines System lokalisiert sind. Repliziert man Software, so repliziert man auch unweigerlich die darin enthaltenen Fehler. Techniken wir Triple Modular Redundancy oder Standby Spares sind bei Softwarefehlern demzufolge nicht sinnvoll einsetzbar. Es gibt aber ähnliche Techniken, die auf *Diversifikation* von Software beruhen, und von denen wir hier Recovery Blocks und N-Versionen-Programmierung kurz beschreiben. Sie basieren auf der Grundidee, verschiedene Versionen von Software zu verwenden. Beide Techniken und ihr Vergleich sind ausführlich in [44, 58] diskutiert.

Recovery Blocks Das Konzept der *Recovery Blocks* basiert auf verschiedenen Versionen von Software und einem sogenannten *Akzeptanztest*. Dieser Test muss in der Lage sein, die Ausgabe einer Komponente auf ihre Korrektheit oder Plausibilität zu überprüfen. Wird das Ergebnis eines Ablaufs für ungültig erklärt, so wird es verworfen, das System in einen Startzustand zurückgesetzt (also ein *recovery* durchgeführt) und eine alternative Version gestartet. Dieser Vorgang läßt sich so lange wiederholen, bis alle Versionen verwendet wurden. In Abbildung 2.8 ist der Ablauf für drei Versionen dargestellt (Die Pfeile beschreiben hier den Kontroll-, nicht den Datenfluß).

Folgende Voraussetzungen ergeben sich für die Anwendung von Recovery Blocks: Es muss einen geeigneten Akzeptanztest geben, der aufgrund des ihm gelieferten Ergebnisses (bei Kenntnis der Eingabedaten) effizient überprüfen kann, ob ein Ergebnis korrekt

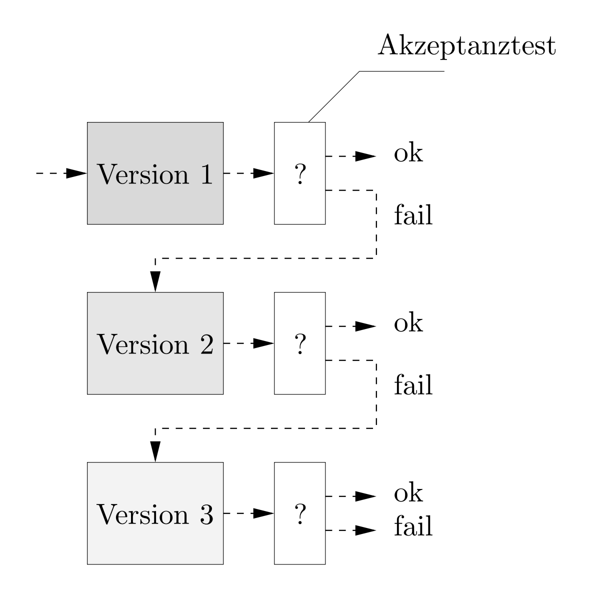


Abbildung 2.8: Recovery Blocks

oder zumindest akzeptabel ist. Bei komplexen Aufgaben kann ein solcher Test nicht existieren oder nur in Form einer aufwendigen Neu- oder Parallelberechnung des Ergebnisses. Ferner muss es möglich sein, das System in seinen Startzustand zurückzuversetzen, wenn ein Ergebnis nicht akzeptabel ist. Bei intensiven Interaktionen mit der Umgebung kann dies eventuell nicht realisiert werden. Bei zeitkritischen Anwendungen müssen Zeitschranken auch dann noch eingehalten werden, wenn erst spät, das heißt im Extremfall im zuletzt gestarteten Modul ein korrektes Ergebnis gefunden wird.

Es ist möglich, die Komplexität der erst später verwendeten Module zu reduzieren, und auf Teile der Funktionalität zu verzichten. Dies stellt eine Form von *graceful degradation* [31] dar.

N-Versionen-Programmierung Diese Technik ist eng verwandt mit der Triple Modular Redundancy. Die verschiedenen Versionen werden parallel ausgeführt und das Ergebnis durch einen *Voter* verglichen, wie in Abbildung 2.9 dargestellt.

Ein geeigneter Voter muss nicht für alle Arten von Software existieren. Es können für eine Berechnung durchaus mehrere Resultate korrekt sein, so dass es durch reine Mehrheitsbildung nicht möglich ist, korrekte von falschen Ergebnissen zu unterscheiden. Unter Umständen können Toleranzen definiert werden, mit denen ähnliche Resultate als Übereinstimmung interpretiert werden.

Die Abarbeitung der verschiedenen Versionen muss effizient und nebenläufig erfolgen können. Bei sequentieller Ausführung müßte die Summe aller Laufzeiten aufgewendet werden; bei paralleler Abarbeitung ergibt sich noch das Maximum der Laufzeiten der Einzelkomponenten. Die Komponenten müssen darüber hinaus vollkommen eigenständig ablaufen können, ohne eine verändernde Wirkung auf ihre Umgebung zu haben. Damit müssen sie eine noch stärkere Eigenschaft erfüllen, als für die Rücksetzbarkeit notwendig ist.

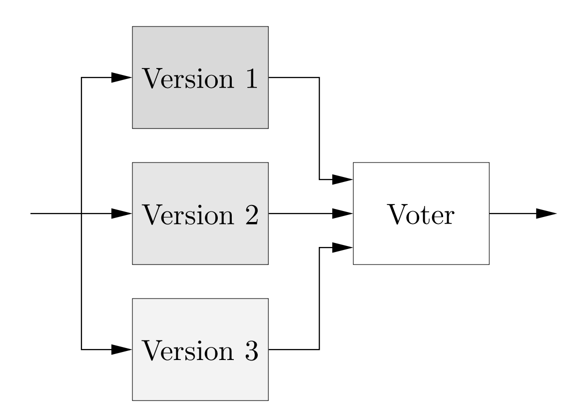


Abbildung 2.9: N-Versionen-Programmierung

Das Problem beider Ansätze ist die Bereitstellung einer „echten“, vielfältigen Diversifikation zwischen den Versionen. Man versucht, dies beispielsweise durch die Verwendung verschiedener Programmierteams, Programmiersprachen und Tools zu erreichen. Dadurch entstehen hohe Qualitätsanforderungen an eine Spezifikation des Systems: Sie muss einerseits sehr präzise sein, damit Versionen entstehen, die nach Möglichkeit alle korrekt sind, und deren Ergebnisse durch den (korrekten) Akzeptanztest angenommen oder die durch einen Voter auch wirklich sinnvoll verglichen werden können. Andererseits soll sie möglichst allgemein und frei von bereits getroffenen Entwurfsentscheidungen sein, um noch verschiedenartige Implementierungen zuzulassen. Eine Diskussion über Techniken und Probleme der Software-Diversifikation mit einem guten Überblick über verschiedene Experimente und ihre Resultate findet sich in [7].

Systeme mit Softwarefehler-Toleranz scheinen nicht sehr häufig entwickelt zu werden und es finden sich kaum Erfahrungsberichte (Storey gibt in [66] einige wenige Verweise). Die Integration genannter Fehlertoleranztechniken zusammen mit der Erstellung mehrerer Versionen resultieren in einem deutlich höheren Entwicklungsaufwand. Dieser Aufwand könnte stattdessen auch in Techniken zur Fehlervermeidung investiert werden, was insbesondere für Softwarefehler interessant ist: Ein einmal korrektes Softwaresystem bleibt für immer korrekt (von Änderungen der Anforderungen oder der Umgebung abgesehen). Die Verwendung von fehlertoleranten Mitteln könnte auch als Eingeständnis interpretiert werden, den Softwareentwicklungsprozeß nicht ausreichend gut zu beherrschen, so dass ein Entwicklungsteam darauf aus Imagegründen bevorzugt verzichtet.

Techniken der Fehlertoleranz werden daher vorwiegend in Systemen zum Einsatz kommen, an die höchste Qualitätsanforderungen gestellt werden. Dabei sollte Fehlertoleranz ein *zusätzliches Mittel* zur Erreichung dieses Zieles sein, aber andere Techniken der Fehlervermeidung keinesfalls ersetzen.

2.3 Zusammenfassung

Im Umgang mit Fehlern lassen sich sehr unterschiedliche Interpretationen der differenzierten Fehlerbegriffe finden. Wir haben die drei wesentliche Begriffe *Fehlerursache*, *Fehlerzustand* und *Versagen* identifiziert und diese informell charakterisiert und klassifiziert. Ein präziser und formaler Umgang mit Fehlern erfordert also ein genaues Verständnis der möglichen Varianten und geeignete Beschreibungstechniken, um mit konkreten Fehlern in konkreten Systemen zielführend umgehen zu können.

Sowohl Fehler als auch Fehlertoleranz sind relative Begriffe: Fehler lassen sich nur relativ zu einer definierten Korrektheit bestimmen. Ein einzelner Fehler kann nur als Abweichung von einem intendierten, erwünschten System, Zustand oder Verhalten sinnvoll angegeben werden. Wir haben eine systematische Darstellung der Fehlerbegriffe vorgestellt, die diese in Abhängigkeit einer Soll-/Ist-Abweichung verschiedener Merkmale eines Systems charakterisiert.

Es existieren formale Ansätze, mit denen Fehler und Fehlertoleranz prinzipiell definiert werden können. Diese bleiben in ihrer Ausdruckskraft aber eingeschränkt und ihre praktische Anwendbarkeit konnte noch nicht nachgewiesen werden. Erweiterungen formaler Ansätze um pragmatische Beschreibungstechniken und methodische Hilfestellungen zur Entwicklung von Systemen unter Einbeziehung der Fehlerproblematik werden sich also gewinnbringend einsetzen lassen.

Es haben sich einige Techniken etabliert, die sich zur Entwicklung fehlertoleranter Systeme eignen, wobei diese vorwiegend auf Hardwarefehler abzielen. In der Beherrschung von Softwarefehlern sind Defizite erkennbar, was seinen Grund nicht zuletzt in der hohen Komplexität von Software hat, die durch die Integration und Modellierung von potentiellen Fehlern und ihren zugehörigen Gegenmaßnahmen noch weiter stark wächst. Wünschenswert ist eine systematische Methodik zum Umgang mit Fehlern, die idealerweise das Normalverhalten von der Fehlerbehandlung trennt.

Mit dieser Arbeit werden wir hierfür einen wesentlichen Beitrag leisten. Wir werden Techniken bereitstellen, mit denen Systeme und ihre Fehler formal und explizit modelliert werden können und werden eine methodische Unterstützung für den integrierten Umgang mit Fehlern im Rahmen einer formalen Systementwicklung erarbeiten.

In den drei folgenden Kapiteln werden wir ein formales Modell einführen und erweitern um Techniken zur Beschreibung von Abweichungen. Wir werden schließlich Techniken vorstellen, die in einer Systementwicklung sinnvoll zum Umgang mit Fehlern verwendet werden können.

Kapitel 3

Formales Systemmodell

Dieses Kapitel beschreibt das Systemmodell der Methodik Focus, das wir als Grundlage für diese Arbeit gewählt haben. Ein Systemmodell bietet eine allgemeine und abstrakte Charakterisierung der Klasse der Systeme, die uns interessieren und sich zur Modellierung verteilter reaktiver Systeme eignen. Es definiert den Systembegriff, die Modellierung von Verhalten, Kommunikation, Komposition und verschiedene Verfeinerungs- und Beweiskonzepte.

Nach einem Überblick über das Systemmodell und einer Darstellung formaler Grundlagen definieren wir die semantische Sicht auf Schnittstellen (Abschnitt 3.3) und auf das Verhalten von Systemen (Abschnitt 3.4). Zur Beschreibung konkreter Systeme führen wir die sogenannten *Black-Box Spezifikationen* und *Zustandstransitionssysteme* in den Abschnitten 3.5 und 3.6 ein. Die Komposition von Systemen wird in 3.7 definiert, gefolgt von Techniken und Begriffen zur *Verifikation* und *Verfeinerung*. Schließlich skizzieren wir einen idealisierten Entwicklungsbegriff und enden mit einem Ausblick auf alternative Systemmodelle. Wir werden zur Illustration der Konzepte eine einfache Komponente *Merge* als durchgängiges Beispiel verwenden. In Kapitel 4 werden wir die Fehlermodellierung auf dem hier vorgestellten Systemmodell abstützen.

Eine detailliertere Beschreibung von Focus findet sich in [21, 19]. Weitere Literatur aus dem Projektumfeld zusammen mit Anwendungen von Focus ist in [25] aufgeführt. Die Beweiskonzepte und -regeln sind ausführlich in [10, 12] vorgestellt, und [10, 11] enthält eine Beschreibung der Verwendung von Verifikationsdiagrammen in Kontext von Focus.

3.1 Überblick

Ein *System* ist – im Sinne von Focus – eine von der Umgebung abgegrenzte Einheit, die durch eine Definition ihrer Schnittstelle zur Umgebung und ihres Verhaltens vollständig charakterisiert werden kann.

Die Schnittstelle besteht aus einer Menge von Kanalnamen. Ein Kanal ist eine Kommunikationsverbindung, über die Nachrichten unidirektional, also in eine Richtung fließen

können. Einem Kanal ist ein Typ zugeordnet als eine Menge von Nachrichten, die auf diesem Kanal gesendet und empfangen werden können.

Das Verhalten wird durch eine Relation zwischen Eingabe und Ausgabe dargestellt. Ein- und Ausgaben werden durch Nachrichtenströme beschrieben, die den Kanälen zugeordnet werden. Ein Strom ist eine (möglicherweise unendliche) Sequenz von Nachrichten, die die gesamte Kommunikationsgeschichte der Lebenszeit eines Systems beschreibt.

Ein System ist typischerweise wieder aus Komponenten zusammengesetzt. Diese Komponenten sind wiederum eigenständige Systeme mit Schnittstelle und Verhalten. Die Kommunikation zwischen den Komponenten findet ausschließlich über den Austausch von Nachrichten über die Kanäle statt.

Für Systeme, ihre Schnittstellen, ihr Verhalten und ihre Komposition sind *Beschreibungstechniken* anzubieten, mit denen konkrete Instanzen beschrieben werden können. Wir geben hierfür verschiedene Techniken an.

Wir werden in der gesamten Arbeit die Begriffe *Systeme* und *Komponenten* synonym verwenden, da es zwischen beiden Begriffen keinen prinzipiellen Unterschied gibt: Ein System kann als Komponente Teil eines umfassenden Systems sein, und eine Komponente ist selbst wieder ein System. Die Wahl der Bezeichnung für ein System bzw. für eine Komponente hängt nur vom Kontext der Verwendung ab und dient bei komponierten Systemen zur besseren sprachlichen Unterscheidung zwischen dem Gesamtsystem und seinen Komponenten.

Wir beginnen den formalen Teil dieses Kapitel mit einer Einführung in benötigte mathematische Grundlagen.

3.2 Mathematische Grundlagen

Die Grundlage zur Darstellung von Kommunikation zwischen Systemen bilden die *Ströme*, die als Sequenz von Nachrichten die Kommunikationsgeschichte auf einem Kanal darstellen können. Wir definieren sie zusammen mit den zugehörigen Operatoren:

Definition 3.1 *Ströme und ihre Operatoren*

Sei M eine beliebige, nichtleere Menge. Ein Strom *über M ist eine endliche oder unendliche Sequenz von Elementen aus M. Die Menge der endlichen bzw. unendlichen Ströme über M wird bezeichnet als M^* bzw. M^∞. Die Menge aller Ströme über M wird definiert als $M^\omega = M^* \cup M^\infty$.*

Für $x_1, \ldots, x_n \in M$ bezeichnet der Ausdruck $\langle x_1, x_2, \ldots, x_n \rangle$ einen endlichen Strom der Länge n mit x_1 als erstem und x_n als letztem Element. Der Sonderfall $\langle\rangle$ bezeichnet den leeren Strom.

$\#x$ bezeichnet die Länge des Stroms x. Ist x unendlich, so gilt $\#x = \infty$.

*Das erste Element eines nicht-leeren Stromes x wird mit $ft.x$ (*first*) bezeichnet, ein Strom x ohne sein erstes Element mit $rt.x$ (*rest*). Das letzte Element eines Stromes wird definiert als $lt.x$ (*last*).*

Falls $t \leq \#x$*, so bezeichnet* $x.t$ *die* t*-te Nachricht in Strom* x*.*

$x \frown y$ *bezeichnet die Konkatenation der Ströme* x *und* y*. Ist* x *ein unendlicher Strom, so gilt* $x \frown y = x$*.*

Der Strom x *ist ein* Präfix *(Anfangsstrom) von* y*, notiert als* $x \sqsubseteq y$*, wenn* x *zu* y *verlängert werden kann:*

$$x \sqsubseteq y \quad \Leftrightarrow_{def} \quad \exists\, z \in M^{\omega} \bullet x \frown z = y$$

x *ist ein* echter Präfix*, also* $x \sqsubset y$*, wenn* y *wirklich länger ist als* x*, also wenn* $z \neq \langle\rangle$*.*

Mit Hilfe des Filter-*Operators* Ⓢ *definiert man mit* DⓈx *den Strom, der aus* x *entsteht, wenn man alle Nachrichten aus* x *entfernt, die nicht in* D *liegen. Er kann definiert werden durch folgende induktive Definition:*

$$D\circledS\langle\rangle = \langle\rangle$$

$$D\circledS(\langle d\rangle \frown x) = \begin{cases} \langle d\rangle \frown (D\circledS x) & \textit{falls } d \in D \\ D\circledS x & \textit{falls } d \notin D \end{cases}$$

□

In [19, 21] finden sich weitere, ausführlich beschriebene Definitionen und Operatoren zu Strömen. Wir haben uns hier auf die Darstellung der Operatoren beschränkt, die in dieser Arbeit verwendet werden.

Da wir Eigenschaften mit Hilfe von Formeln definieren wollen, bedarf es entsprechender Begriffe, die wie üblich charakterisiert werden:

Definition 3.2 *Variable, Belegungen, Formeln*

Die Menge VAR bezeichnet die Menge aller Variablen. Jeder Variable $v \in VAR$ *ist ein* Typ *zugewiesen. Ein Typ ist eine Menge von Werten, die eine Variable annehmen kann.*

Eine Belegung α *weist jeder Variablen* v *einen Wert* $\alpha.v$ *ihres Typs zu. Die Menge aller Belegungen wird durch VAL bezeichnet.*

Zwei Belegungen stimmen auf einer Variablenmenge V *überein, wenn sie allen Variablen aus* V *jeweils den gleichen Wert zuweisen:*

$$\alpha \stackrel{V}{=} \beta \quad \stackrel{df}{=} \quad \forall\, v \in V \bullet \alpha.v = \beta.v$$

Zu zwei Belegungen bezeichnet $Agree(\alpha, \beta)$ *(für* agreement*) die maximale Menge von Variablen, auf denen die Belegungen* α *und* β *übereinstimmen:*

$$Agree(\alpha, \beta) \subseteq VAR$$

mit

$$\alpha \stackrel{Agree(\alpha,\beta)}{=} \beta \wedge \forall\, V' \subseteq VAR \bullet \alpha \stackrel{V'}{=} \beta \Rightarrow V' \subseteq Agree(\alpha, \beta)$$

Eine Formel Φ *ist ein (wohlgeformter) Ausdruck, der sich mit einer Belegung* α *zur Interpretation der freien Variablen zu einem booleschen Wahrheitswert aus* $\{\mathsf{true}, \mathsf{false}\}$

auswerten läßt. Wird Φ *unter* α *zu* `true` *ausgewertet, notieren wir dies als*

$$\alpha \models \Phi$$

Für eine Variablenmenge V *bezeichnet* V' *die zu* V *disjunkte Variablenmenge, die alle Variable aus* V *mit einem Strich versieht, also*

$$V' = \{v' \mid v \in V\}$$

Dabei stellen v *und* v' *verschiedene Variablen dar. Für eine Belegung* α *kann eine Belegung* α' *definiert werden. Diese weist einer Variablen* v' *denselben Wert zu, der von* α *an* v *zugewiesen wird, also*

$$\forall\, v \in V \bullet \alpha.v = \alpha'.v'$$

Um einer Formel, die sowohl einfache als auch gestrichene Variablen enthält, einen Wahrheitswert zuzuweisen, verwenden wir eine Belegung α *und eine gestrichene Belegung* β'*. Die Aussage*

$$\alpha, \beta' \models \Phi$$

gilt, wenn Φ *wahr wird, wobei die einfachen Variablen mittels* α*, die gestrichenen Variablen mittels* β *ausgewertet werden.*

Die Menge der freien Variablen in Φ *wird bezeichnet als* $free(\Phi)$*. Mit* $x \in free(\Phi)$ *beschreibt* $\Phi[y/x]$ *die Formel* Φ*, in der aber alle Vorkommen von* x *durch* y *ersetzt sind.* □

3.3 Systemschnittstelle

Die *Schnittstelle* eines Systems beschreibt seine Ein- und Ausgabekanäle. Sie wird definiert durch zwei Mengen, die jeweils die Namen der Kanäle angeben. Für die Kanäle müssen zugehörige Typen definiert sein.

Definition 3.3 *Schnittstelle*

Die Schnittstelle eines Systems wird durch ein Paar (I, O)*, bestehend aus einer endlichen Menge* I *der Eingabekanäle sowie einer endlichen Menge* O *der Ausgabekanäle beschrieben (mit* $I \cup O \subset VAR$*). Allen Kanälen* $c \in I \cup O$ *werden durch die Funktion* type *Typen zugeordnet, die die Menge der möglichen Nachrichten enthalten, die über einen Kanal gesendet und empfangen werden können.* □

Die Mengen I und O werden typischerweise graphisch repräsentiert, wie dies in Abbildung 3.1 für das folgende Beispiel dargestellt ist.

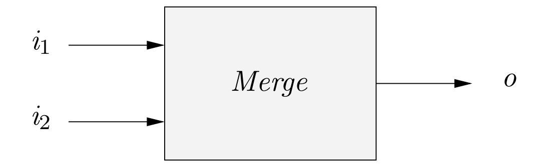


Abbildung 3.1: Schnittstelle von *Merge*

Beispiel 3.1 Wir betrachten in diesem und auch dem nächsten Kapitel ein einfaches System namens *Merge*, an dem sich die Konzepte dieser Arbeit gut veranschaulichen lassen. Das System soll - ähnlich einem Multiplexer - die Nachrichten, die es über zwei Kanäle empfängt, auf seinem Ausgabekanal wieder ausgeben.

Das System hat also zwei Eingabekanäle i_1 und i_2. Über den Kanal i_1 können Nachrichten eines Typs D_1, über i_2 Nachrichten vom Typ D_2 empfangen werden. Die beiden Nachrichtenmengen seien nicht weiter spezifiziert. Für sie sei lediglich angenommen, dass sie disjunkt sind, also $D_1 \cap D_2 = \varnothing$ gilt. Auf dem Ausgabekanal o können alle empfangenen Nachrichten wieder ausgegeben werden, wodurch sich der Typ von o als die Vereinigung aller Nachrichten von D_1 und D_2 ergibt. Formal definieren wir also

$$I = \{i_1, i_2\} \qquad O = \{o\}$$
$$type(i_1) = D_1 \qquad type(i_2) = D_2 \qquad type(o) = D_1 \cup D_2$$

$\square$

In dem hier verwendeten Systemmodell ist die Schnittstelle eines Systems (bzw. einer Komponente) statisch, bleibt also (nach der Implementierung) unverändert. Systeme mit Komponenten, deren Schnittstellen sich zur Laufzeit ändern, werden in der vorliegenden Arbeit nicht betrachtet. Das derart verallgemeinerte *mobile, dynamische* FOCUS wird in [29, 30, 32] beschrieben .

3.4 Systemverhalten

Das Verhalten eines Systems wird charakterisiert durch die Beziehung zwischen den *Eingaben* an das System und den *Ausgaben*, mit denen das System auf erstere reagiert. Um ein konkretes Verhalten anzugeben, müssen wir also für jede mögliche Eingabe definieren, mit welcher Ausgabe das System darauf reagieren wird.

Im allgemeinen hängt diese Reaktion nicht nur von einer einzelnen Nachricht in der Eingabe zu einem bestimmten Zeitpunkt ab, sondern auch von der gesamten Vorgeschichte, also den zuvor bereits empfangenen und verarbeiteten Nachrichten. So hängt beispielsweise die Reaktion eines Lesebefehls an einen Speicher davon ab, welche Daten vorher in den Speicher geschrieben wurden. Eine derartige Information wird üblicherweise in einem *Zustand* des Systems gespeichert. Da wir Zustände eines Systems aber (vorerst) nicht modellieren wollen, setzen wir komplette Eingabe*geschichten* mit Ausgabe*geschichten* in Beziehung. Die Präfixe der Kommunikationsgeschichten lassen sich als abstrakte Repräsentationen von Zuständen interpretieren.

Formal können wir die Beziehung von Ein- und Ausgabegeschichten ausdrücken durch eine Relation R, die ganze Eingabeströme und Ausgabeströme zueinander in Beziehung

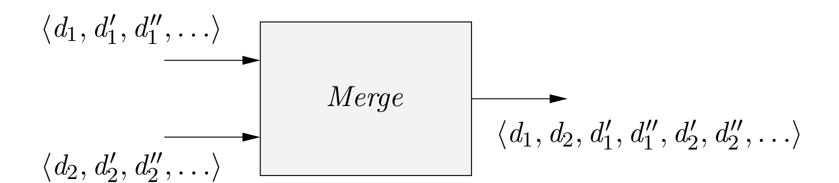


Abbildung 3.2: Verhalten von *Merge* (Beispiel)

setzt. Mit n Eingabekanälen $\{i_1, \ldots, i_n\}$ und m Ausgabekanälen $\{o_1, \ldots, o_m\}$ eines Systems ergibt sich R formal als eine Relation vom Typ

$$R \quad \subseteq \quad type(i_1)^\omega \times \ldots type(i_n)^\omega \times type(o_1)^\omega \times \ldots type(o_m)^\omega$$

Diese Sicht auf Systeme wird *Black-Box*-Sicht genannt, da nur das nach außen sichtbare Kommunikationsverhalten beschrieben wird, ohne Bezug auf Interna des Systems wie lokale Variable, Zustände oder interne Operationen.

Wir geben das Verhalten von *Merge* im folgenden Beispiel durch eine Relation an.

Beispiel 3.2 Das Verhalten von *Merge* sei zunächst informell beschrieben: Die Nachrichten, die auf i_1 und i_2 empfangen werden, sollen auf o wieder ausgegeben werden. Die Reihenfolge der Daten soll dabei nicht verändert werden: Wenn also eine Nachricht d_j vor einer Nachricht d_j' auf Kanal i_j empfangen wird, dann soll auch auf o wieder d_j vor d_j' gesendet werden (für $j \in \{1, 2\}$). Die Art und Weise der Zusammenmischung von Nachrichten aus *verschiedenen* Kanälen bleibt dabei beliebig, d.h. ohne Vorgabe von Reihenfolgen oder Prioritäten.

Dieses Verhalten von *Merge*, dessen informelle Beschreibung trotz seiner Einfachheit bereits relativ aufwendig ist, wird formal spezifiziert durch eine Relation, die wir auch mit dem Bezeichner *Merge* identifizieren:

$$Merge \subseteq D_1^\omega \times D_2^\omega \times (D_1 \cup D_2)^\omega$$

Sie ist beispielhaft angedeutet in Abbildung 3.2 und durch folgende explizite, aber nur partielle Angabe. Dabei seien $d_1, d_1', \ldots$ Elemente aus D_1, und $d_2, d_2', \ldots$ Nachrichten aus D_2.

$$
\begin{array}{llll}
Merge = \{ & (\langle\rangle, & \langle\rangle, & \langle\rangle), \\
& (\langle d_1\rangle, & \langle\rangle, & \langle d_1\rangle), \\
& (\langle\rangle, & \langle d_2\rangle, & \langle d_2\rangle), \\
& (\langle d_1, d_1'\rangle, & \langle\rangle, & \langle d_1, d_1'\rangle), \\
& (\langle d_1\rangle, & \langle d_2\rangle, & \langle d_1, d_2\rangle), \\
& (\langle d_1\rangle, & \langle d_2\rangle, & \langle d_2, d_1\rangle), \\
& (\langle\rangle, & \langle d_2, d_2'\rangle, & \langle d_2, d_2'\rangle), \\
& (\langle d_1, d_1'\rangle, & \langle d_2\rangle, & \langle d_2, d_1, d_1'\rangle), \\
& (\langle d_1, d_1'\rangle, & \langle d_2\rangle, & \langle d_1, d_2, d_1'\rangle), \\
& (\langle d_1, d_1'\rangle, & \langle d_2\rangle, & \langle d_1, d_1', d_2\rangle), \\
& \ldots & & \qquad\qquad \}
\end{array}
$$

Für die Eingabe $\langle d_1, d_1'\rangle$ auf i_1 und $\langle d_2\rangle$ auf i_2 gibt es beispielsweise drei mögliche Reaktionen im Sinne der informellen Beschreibung, da die Nachricht d_2 vor, zwischen oder nach den beiden Nachrichten von i_1 wieder auf o ausgegeben werden kann.

Selbstverständlich ist es im allgemeinen nicht möglich, alle potentiellen Verhalten explizit aufzulisten, da die Relation unendlich viele Elemente enthält und sogar einzelne Elemente unendlich lange Ströme enthalten können. Die Relation muss durch geeignete Beschreibungstechniken angegeben werden. □

Mit Relationen können Paarungen von Ein- und Ausgaben ausgedrückt werden, die einem nicht mehr sinnvollen, also kausalem und realisierbarem Verhalten entsprechen. Ein Verhalten ist realisierbar, wenn es eine *Strategie* gibt, die Eingaben an ein System schrittweise liest und aus diesen die Ausgaben produziert. Dabei darf die Reaktion nicht von Nachrichten abhängen, die zu einem aktuellen Verarbeitungszeitpunkt noch gar nicht verfügbar sind. Dies widerspräche der Kausalität, und käme der Fähigkeit des Systems gleich, in die Zukunft sehen zu können. In [21] wird die Realisierbarkeit genauer im Rahmen eines gezeiteten Systemmodells diskutiert und begründet, dass die Realisierbarkeit im wesentlichen nur durch die Verletzung der Linkstotalität und der Kausalität gefährdet werden kann.

Um die Realisierbarkeit von Systemen in unserem Systemmodell sicherzustellen, fordern wir von den Relationen die folgenden zwei Eigenschaften:

Linkstotalität Ein System zeigt für jedes Verhalten der Umgebung – also für jede Eingabe – eine Reaktion. Die verhaltensbeschreibende Relation muss also für jede mögliche Eingabe ein passendes Element besitzen, formal

$$\forall\, i \bullet \exists\, o \bullet (i, o) \in R$$

Monotonie Ein System darf eine einmal ausgesandte Nachricht nicht mehr zurücknehmen, d.h. verlängert man eine Eingabe i zu i', so verlängert sich auch die Ausgabe (oder bleibt unverändert):

$$\forall\, i, i', o, o' \bullet (i, o) \in R \wedge i \sqsubseteq i' \wedge (i', o') \in R \Rightarrow o \sqsubseteq o'$$

In unserem ungezeiteten Modell stellt dies auch die Kausalität sicher: Haben zwei Eingaben einen gemeinsamen Präfix i, so haben auch ihre Ausgaben einen gemeinsamen Präfix o, für den auch $(i, o) \in R$ gilt. Dies impliziert, dass die Reaktion auf einen Teil der Eingaben wirklich nur durch diesen Teil, und nicht durch erst noch zu empfangene Nachrichten bestimmt wird.

In der Praxis ist ein Nachweis dieser Eigenschaften oft nicht erforderlich, da durch die Beschreibungstechniken ihre Erfüllung automatisch sichergestellt ist. So stellen beispielsweise die in Abschnitt 3.6 beschriebenen Transitionssysteme durch ihre schrittweise Verlängerung der Ausgaben die Monotonie sicher.

Um die Lesbarkeit zu erhöhen, haben wir die beiden obigen Forderungen in einer vereinfachten Notation ausgedrückt, die wir im Verlauf der Arbeit noch oft verwenden werden:

Notationelle Konvention Wir gehen für eine vereinfachte Notation für Charakterisierungen von Verhaltensrelationen von nur *einem* Eingabekanal und *einem* Ausgabekanal aus, und identifizieren den Bezeichner für einen Kanal mit dem Strom, der auf ihm übertragen wird.

Wir wählen also für die Notation für die Ströme auf dem Eingabekanal die Bezeichnung i, und für Ströme auf dem Ausgabekanal o. Damit vereinfacht sich beispielsweise der Ausdruck

$$\forall\, x_1 \in \mathit{type}(i_1)^\omega, \ldots, x_n \in \mathit{type}(i_n)^\omega \bullet \ldots$$

zu einem kurzen

$$\forall\, i \bullet \ldots$$

Die Forderung der Linkstotalität müßte ausführlich notiert werden als

$$\begin{array}{l} \forall\, x_1 \in \mathit{type}(i_1)^\omega, \ldots, x_n \in \mathit{type}(i_n)^\omega \bullet \\ \quad \exists\, y_1 \in \mathit{type}(o_1)^\omega, \ldots, y_m \in \mathit{type}(o_m)^\omega \bullet \\ \qquad (x_1, \ldots, x_n, y_1, \ldots, y_m) \in R \end{array}$$

Wie im Beispiel bereits angedeutet, enthält eine verhaltensbeschreibende Relation im allgemeinen unendlich viele Tupel, wobei bereits ein einzelnes Tupel unendlich lange Ströme enthalten kann. Daher kann eine konkrete Relation nicht durch explizite Auflistung angegeben werden, was aber auch im Fall von endlichen Relationen keine angemessene Technik wäre. In den nächsten Abschnitten werden wir Formeln (Abschnitt 3.5) und Diagramme (Abschnitt 3.6) als geeignetere Beschreibungstechniken angeben.

3.5 Black-Box Spezifikationen

Wie wir im vorherigen Abschnitt gesehen haben, läßt sich das Verhalten eines Systems abstrakt als eine Relation über Strömen betrachten. Typischerweise beschreibt man eine Relation durch eine Formel, die die Elemente charakterisiert, die in der Relation enthalten sind. Diese Beschreibungstechnik verwenden wir auch für unsere Verhaltensspezifikationen.

Eine *Black-Box Spezifikation* beschreibt das Verhalten eines Systems durch ein Prädikat. Wir gehen von einem System aus, für das die Schnittstelle bereits definiert sei. In dem Prädikat dürfen die Bezeichner für die Kanäle des Systems als freie Variable vorkommen. Das Prädikat charakterisiert dann diejenigen Tupel von Ein- und Ausgabeströmen, für die das Prädikat wahr wird, wenn man die Kanalbezeichner durch die Ströme ersetzt, die auf ihnen empfangen bzw. gesendet werden.

Definition 3.4 *Black-Box Spezifikation*

Eine Black-Box Spezifikation eines Systems mit der Schnittstelle (I, O) *besteht aus einem Prädikat* Φ*, das Variablen aus* $I \cup O$ *als freie Variablen hat, d.h.* $\mathit{free}(\Phi) \subseteq I \cup O$

Das Prädikat beschreibt die Menge aller Ein- und Ausgabeströme, die dieses Prädikat erfüllen:

$$[\![\phi]\!] \stackrel{df}{=} \{\ (i_1, \ldots, i_n, o_1, \ldots, o_m) \ \mid\ \phi\ \}$$

□

Es ist möglich, ein Prädikat zu definieren, das nicht konsistent, nicht linkstotal oder auch nicht kausal ist. Das Prädikat beschreibt in einem solchen Fall kein realisierbares Verhalten, und es gibt kein konkretes, implementiertes System, dessen Ein- und Ausgabeströme das Prädikat erfüllen könnten. Die Konsistenz und Kausalität muss bei Bedarf durch Nachweis sichergestellt werden. Im Rahmen einer Systementwicklung kann es allerdings ausreichend sein, dies erst indirekt durch die Entwicklung einer Implementierung zu tun.

Beispiel 3.3 Das System *Merge* kann sehr leicht in der Black-Box Sicht spezifiziert werden. Betrachtet man im Ausgabestrom o die Nachrichten, die vom Eingabekanal i_1 kommen können, so sind dies genau die Nachrichten, die auf i_1 tatsächlich empfangen wurden, und sie erscheinen in unveränderter Reihenfolge. Extrahiert man also aus o den Strom der Nachrichten vom Typ D_1, so ist dieser identisch mit i_1. Die analoge Forderung für i_2 ergibt die vollständige Black-Box Spezifikation. Wir benennen das Prädikat entsprechend dem Namen des Systems:

$$\mathit{Merge}^{\Phi} \stackrel{df}{=} D_1 Ⓢ o = i_1 \wedge D_2 Ⓢ o = i_2$$

Die Stromtupel aus Beispiel 3.2 erfüllen alle die hier genannte Spezifikation. □

Eine spezielle Form von Black-Box Spezifikationen stellen die sogenannten *Assumption/Guarantee* (oder auch *Rely/Guarantee* oder *Assumption/Commitment*) Spezifikationen dar, die die Funktionalität eines Systems in Abhängigkeit von Eigenschaften der Umgebung definieren.

Einfache Assumption/Guarantee-Spezifikationen

Eine einfache A/G-Spezifikation besteht aus zwei Prädikaten A und G. Das Prädikat A weist als freie Variable nur Kanäle aus I auf und beschreibt, welche Eingabeströme von der Umgebung an das System geschickt werden dürfen. Darin ist also die *Annahme* an die Umgebung formuliert, deren Gültigkeit vorausgesetzt wird. Mit dem Prädikat G wird die Verhaltensrelation des Systems auf die gleiche Weise definiert wie für einfache Black-Box Spezifikationen.

Als Semantik einer einfachen A/G-Spezifikation definieren wir also (unter Verwendung der vereinfachten Notation entsprechend Seite 43) alle Stromtupel, die G erfüllen, wenn die Umgebung nur Eingabeströme sendet, die A erfüllen:

$$[\![(A, G)]\!] \stackrel{df}{=} \{(i, o) \mid A(i) \Rightarrow G(i, o)\}$$

Erfüllt eine Umgebung die Annahme A nicht, ist das Verhalten des Systems unspezifiziert, also beliebig.

Normalerweise erbringt ein System seine Leistung schrittweise, indem es schrittweise Nachrichten einliest, auf diese mit Ausgaben reagiert, weiter Eingaben liest, wieder entsprechende Ausgaben macht und so fort. Solange die Umgebung sich an A hält, muss sich auch das System an G halten. Obige Definition scheint es zuzulassen, dass sich ein System von Anfang an beliebig verhält, auch wenn eine Abweichung von der Annahme erst bei einer späteren Nachricht im Eingabestrom i auftritt: Für dieses i

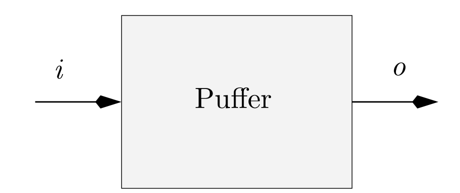


Abbildung 3.3: Schnittstelle von *Buffer*

wäre A nicht erfüllt, und die Implikation für ein beliebiges o aber schon. Durch die geforderte Monotonie ist jedoch sichergestellt, dass für einen korrekten Präfix von i auch eine korrekte Ausgabe produziert wird, die wiederum ein Präfix von o ist. Bevor einem System beliebiges Verhalten erlaubt ist, muss die Umgebung also die Annahme A verletzen.

A/G-Spezifikationen sind gut geeignet, die Linkstotalität von Black-Box Spezifikationen zu erreichen, indem nicht-zugelassene Eingaben durch die Annahme ausgeschlossen werden können. Sie eignen sich auch, die Anforderungen an eine Umgebung auszudrücken, in denen Komponenten in der Lage sind, ihre Leistung zu erbringen. Es können damit auch Bedingungen für die Komposition von Komponenten präzise definiert werden.

Beispiel 3.4 A/G-Spezifikation eines Puffers

Als Beispiel spezifizieren wir einen einfachen Puffer, der Daten aus einer (nicht näher spezifizierten) Menge D zwischenspeichert und auf Anfrage in der gleichen Reihenfolge wieder ausgibt. Seine Kapazität sei auf N Datenelemente beschränkt. Es gibt zwei Fälle, die wir durch eine geeignete Umgebungsannahme A ausschließen wollen: Der Puffer empfängt Anfragen, obwohl er keine Daten gespeichert hat, oder der Puffer empfängt Daten, obwohl bereits N Elemente gespeichert sind.

Zunächst definieren wir die Schnittstelle. Sowohl Daten als auch Anfragen (dargestellt durch die Nachricht R) sollen beide auf einem Kanal i empfangen werden, während die Ausgaben auf dem Kanal o wieder ausgegeben werden. Damit ergibt sich

$$\begin{array}{lllllll} I & = & \{i\} & type(i) & = & D \cup \{R\} \\ O & = & \{o\} & type(o) & = & D \end{array}$$

und eine Darstellung der Schnittstelle entsprechend Abbildung 3.3.

Die Annahme A soll die Bedingung beschreiben, die der Eingabestrom zu erfüllen hat. Es ergeben sich folgende beiden Forderungen:

1. Mit jedem Empfang eines Datums verringert sich die Kapazität des Puffers um 1, mit jeder Anfrage und dadurch ausgelöster Ausgabe eines Datums erhöht sich die Kapazität wieder um 1. Die Differenz der empfangenen Daten und der empfangenen Anfragen darf also niemals höher als N liegen; nur dann würde der Puffer seine Kapazitätsgrenze überschreiten. Es muss also gelten

   $$\#D \circledS i - \#\{R\} \circledS i \leq N$$

   Diese Forderung ist allerdings noch zu schwach, denn sie macht nur eine Aussage über die gesamte, möglicherweise unendliche Kommunikationsgeschichte. Die Forderung muß aber

während des gesamten Ablaufes, also auch in allen Zwischen„zuständen“ gelten. Dies wird ausgedrückt, indem die Gültigkeit der obigen Bedingung für alle Präfixe von i gefordert wird:

$$\forall\, i' \sqsubseteq i \bullet \#D \circledS i' - \#\{R\} \circledS i' \leq N$$

2. Zu vermeiden ist der Fall, dass der Puffer eine Anfrage in einer Situation erhält, in der überhaupt keine Daten im Puffer enthalten sind. Dies passiert, wenn mehr Anfragen als Daten empfangen werden, wenn also obige Differenz negativ wird. Um dies auszuschließen, gehen wir davon aus, dass dieser Fall nie, also für keinen Präfix von i eintritt und fordern entsprechend

$$\forall\, i' \sqsubseteq i \bullet \#D \circledS i' - \#\{R\} \circledS i' \geq 0$$

Die Konjunktion dieser beiden Forderungen ergibt unsere Umgebungsannahme. Unter dieser Annahme ist nun das Verhalten des Puffers als Prädikat G zu formulieren.

Der Inhalt der Ausgabe auf dem Kanal o wird in Abhängigkeit von i ausgedrückt. Da gemäß der informellen Spezifikation die Daten auf o genau die empfangenen Daten von i sind, wobei auch die Reihenfolge der Daten erhalten bleibt, fordern wir

$$o \sqsubseteq D \circledS i$$

Diese Sicherheitseigenschaft läßt es zu, dass ein System überhaupt keine Ausgaben macht. Wenn es aber Ausgaben macht, dann sind es die richtigen. Wir benötigen zusätzlich noch eine Lebendigkeitsaussage, die fordert, dass Ausgaben tatsächlich gemacht werden. Dies können wir über die Länge des Stromes o formulieren. Für jede Anfrage wird genau ein Datum auf o ausgegeben:

$$\#o = \#\{R\} \circledS i$$

Beide Aussagen können wir nun zusammenfassen zur A/G-Spezifikation (A, G) des Puffers mit

$$\begin{array}{lcl} A & \stackrel{df}{=} & \forall\, i' \sqsubseteq i \bullet 0 \leq \#D \circledS i' - \#\{R\} \circledS i' \leq N \\ G & \stackrel{df}{=} & o \sqsubseteq D \circledS i \quad \wedge \quad \#o = \#\{R\} \circledS i \end{array}$$

Das Format von A/G-Spezifikationen schränkt das Verhalten nur ein für den Fall, dass die Annahme A erfüllt ist. Gilt die Annahme für einen Eingabestrom i nicht, so ist auch das resultierende Verhalten des Puffers nicht spezifiziert. In Abschnitt 5.6 werden wir den Puffer weiterentwickeln, so dass auch diese Fälle berücksichtigt werden und der Puffer dann ein definiertes und sinnvolles Verhalten zeigt. □

Allgemeine Assumption/Guarantee-Spezifikationen

Im allgemeinen sind die einfachen A/G-Spezifikationen nicht ausdrucksstark genug. Es ist möglich, dass eine Bedingung an die Umgebung nicht mit einem Prädikat definiert werden kann, das nur auf die Eingabekanäle Bezug nimmt. Zu seiner Formulierung können auch Information über die Ausgaben des Systems notwendig werden. Dies kann bei Systemen auftreten, bei denen die Umgebung Bedingungen erfüllen muss, die von einer nichtdeterministischen Wahl der Ausgaben des Systems abhängen. Dann muss auch die Formulierung der Umgebungsannahme Bezug auf die Ausgaben nehmen können. Eine Annahme A hat dann also alle Kanalnamen $I \cup O$ als freie Variable.

Da wir allgemeine A/G-Spezifikationen in dieser Arbeit nicht benötigen, werden wir uns auf einfache A/G-Spezifikationen beschränken und verweisen auf die ausführliche Behandlung dieses Spezifikationstypus in [16, 21, 64, 65].

3.6 Zustandstransitionssysteme

Während Black-Box Spezifikationen das Verhalten von Systemen durch die Relation von *vollständigen* Kommunikationsgeschichten der Ein- und Ausgabe beschreiben, wird mit (Zustands-) Transitionssystemen angegeben, wie das Verhalten *schrittweise* erzeugt wird. Ein Schritt besteht aus dem Übergang von einem *Zustand* des Systems zu einem Nachfolgezustand, wobei Nachrichten von den Eingabekanälen gelesen und auf den Ausgabekanälen geschrieben werden können.

Zustandstransitionssysteme sind durch das schrittweise Erbringen ihrer Ausgaben als Reaktion auf jeweils aktuelle Eingaben unter Berücksichtigung eines Systemzustandes sehr nahe an konkreten Systemimplementierungen. Während wir die Verwendung von Black-Box Spezifikationen als *eigenschaftsorientierte* Spezifikationstechnik bezeichnen können, lassen sich Transitionssysteme als *abstrakte Implementierungen* begreifen. Transitionssysteme für unser Systemmodell wurden in ähnlicher Form in [10] präsentiert. Die Transitionen werden hier allerdings nicht durch Formeln, sondern spezifischer durch Paare von Belegungen definiert. Wir motivieren die folgenden Forderungen, die an ein Transitionssystem gestellt werden, im Anschluß an die Definition.

Definition 3.5 *Transitionssystem*

Ein Zustandstransitionssystem $\mathcal{S}$ wird definiert durch das Tupel

$$\mathcal{S} = (I, O, A, \Upsilon, T)$$

das die folgenden Bedingungen erfüllt. Die Mengen I und O enthalten die Namen der Ein- und Ausgabekanäle, und definieren damit die Schnittstelle des Systems. Die Menge A definiert die zusätzlichen lokalen Variablen des Systems. Es gelte:

(a) $\{i^\circ \mid i \in I\} \subseteq A$

Die Menge der Startzustände wird durch die Formel Υ charakterisiert ($free(\Upsilon) \subseteq I \cup O \cup A$). Das System hat in einem Zustand zu starten, in dem Υ erfüllt ist. Es gelte (für alle $i \in I$)

(b) $\alpha \models \Upsilon \wedge i \in I \Rightarrow \alpha.i^\circ = \langle\rangle$

(c) $\alpha \models \Upsilon \wedge \beta \overset{O \cup A}{=} \alpha \Rightarrow \beta \models \Upsilon$

Die Menge $T \subseteq VAL \times VAL$ enthält die Menge der Transitionen. Sie hat (für alle $i \in I$) folgende Bedingungen zu erfüllen:

(d)
$$\begin{aligned}(\alpha, \beta) \in T \;\Rightarrow\; & \alpha.i^\circ \sqsubseteq \beta.i^\circ \;\wedge\; \beta.i^\circ \sqsubseteq \alpha.i \;\wedge\\ & \alpha.o \sqsubseteq \beta.o \;\wedge\; \alpha.i \sqsubseteq \beta.i\end{aligned}$$

(e) $(\alpha, \beta) \in T \wedge \alpha.i \sqsubseteq \gamma.i \wedge \beta \overset{O \cup A}{=} \gamma \;\Rightarrow\; (\alpha, \gamma) \in T$

Die Menge der Umgebungstransitionen T^ϵ eines Systems wird definiert durch

$$T^\epsilon = \{(\alpha, \beta) \mid \alpha \overset{O \cup A}{=} \beta \wedge \forall\, i \in I \bullet \alpha.i \sqsubseteq \beta.i\}$$

□

Wir wollen die in der Definition auftretenden Einschränkungen erläutern. Die Menge A definiert die zusätzlichen Variablen, die vom System kontrolliert werden können. Typischerweise enthält sie eine Variable für einen Kontrollzustand und Variablen zum Speichern von Daten. Gemäß (a) wird für jeden Eingabestrom i eine Variable i° aufgenommen. Diese beschreibt den Teil der Eingabe, der bereits vom Eingabestrom i konsumiert wurde, also einen endlichen Präfix von i. Bei Start des Systems wurden noch keine Eingaben konsumiert, gefordert in (b). Die ersten beiden Glieder der Konjunktion in (d) stellen weitere Forderungen sicher: Da das Lesen von Nachrichten nicht rückgängig gemacht werden kann, kann bei jedem Schritt (also für jede Transition) der konsumierte Teil nur länger werden, und es können nur Nachrichten gelesen werden, die tatsächlich auf dem Kanal liegen. Wir fordern nicht, dass jedes System mit leeren Ausgabekanälen starten muss, so dass spontane initiale Ausgaben spezifiziert werden können.

Da die Eingabe an ein System von diesem nicht kontrolliert werden kann, darf auch die Initialbedingung die Belegung der Eingabekanäle nicht einschränken. Bedingung (c) stellt sicher, dass beliebige Eingaben möglich sind, und nur Anforderungen an die *kontrollierten* Variablen in $O \cup A$ gestellt werden.

Wir lassen weiterhin nur Systeme zu, deren Transitionen sicherstellen, dass Ausgaben des Systems nicht zurückgenommen werden, und dass auch die Umgebung dies nicht mit ihren bereits gesendeten Nachrichten tun kann. Für alle Transitionen dürfen die Belegungen der i und o also nur länger werden, wie in der zweiten Zeile in (d) formalisiert. Schließlich dürfen die Transitionen des Systems nicht das Verhalten der Umgebung einschränken – gefordert in (e): Ein Berechnungsschritt darf nicht verbunden sein mit einer spezifischen Verlängerung der Eingabe, sondern muss beliebige Verlängerungen zulassen. Entsprechende Transitionen müssen also in T mit enthalten sein.

Für alle Eingabeströme gilt jederzeit $i^\circ \sqsubseteq i$, wie durch Induktion leicht zu zeigen ist: Initial gilt mit $i^\circ = \langle\rangle$ die Aussage trivial, und für jeden Schritt mit $(\alpha, \beta) \in T$ gilt aufgrund von (d) sofort $\beta.i^\circ \sqsubseteq \alpha.i \sqsubseteq \beta.i$. Wir können daher den *ungelesenen* Teil eines Eingabestromes definieren als i^+ durch die Forderung

$$i = i^\circ \frown i^+$$

Die Umgebungstransitionen lassen alle kontrollierten Variablen eines Systems unverändert, können jedoch die Eingabeströme beliebig verlängern. Damit kann ein System für jeden Eingabekanal jeden beliebigen Eingabestrom erzeugen. Damit wird sichergestellt, dass in der Menge der Abläufe (vgl. Definition 3.7) wirklich für alle Eingaben Reaktionen definiert werden.

Ein *Zustand* eines Transitionssystems besteht aus einer Belegung $\alpha \in VAL$, die den Variablen aus $I \cup O \cup A$ Werte zuweist. Die Darstellung eines Zustandes ist nicht eindeutig, da es beliebig viele andere Belegungen gibt, die auf den (endlich vielen) Variablen des Systems übereinstimmen, aber anderen Variablen beliebige andere Werte zuweisen. Die

relevanten Variablen eines Systems bezeichnen wir mit

$$V \stackrel{df}{=} I \cup O \cup A$$

Beschreibung konkreter Transitionssysteme Wie schon bei den Black-Box Spezifikationen kann die meist unendliche Menge T nicht explizit durch Auflistung angegeben werden. Sie wird meist durch eine Menge von Prädikaten τ angegeben, die jeweils ganze Teilmengen von T beschreiben. Ein Prädikat τ enthält freie Variablen aus $V \cup V'$. Die Variablen aus V beschreiben die Werte der Variablen in einem Zustand α, die „gestrichenen" Variablen aus V' die Werte im Nachfolgezustand β. Einem Prädikat τ wird also die Menge von Transitionen

$$\{(\alpha, \beta) \mid \alpha, \beta' \models \tau\}$$

zugeordnet, und einer Menge von Prädikaten τ wird erwartungsgemäß die Vereinigung der zugehörigen Transitionsmengen zugeordnet. Wir werden im folgenden auch ein Prädikat τ als Transition bezeichnen, obwohl dies eigentlich einer Menge von Transitionen entspricht.

Wir können nun definieren, wann eine Transition in einem Systemzustand schaltbereit ist:

Definition 3.6 *Schaltbereitschaft*

Sei ein Transitionssystem (I, O, A, Υ, T) gegeben.

- *Eine (Einzel-) Transition (β, γ) aus T heißt* schaltbereit *in einem Zustand α, wenn sich β von α nur in der Zuweisung von Werten an Variable unterscheidet, die im System nicht auftreten, wenn also gilt:*

  $$\alpha \stackrel{V}{=} \beta$$

- *Eine Transition(smenge) τ ist schaltbereit in einem Zustand α, wenn es einen Nachfolgezustand gibt, der von α aus mit τ erreichbar ist:*

  $$\exists \beta \bullet \alpha, \beta' \models \tau$$

  Dies wird abgekürzt durch

  $$\alpha \models \mathsf{En}(\tau)$$ □

Wir veranschaulichen die Formalisierung von Transitionssystemen anhand des folgenden Beispiels.

Beispiel 3.5 *Merge* als Zustandstransitionssystem

Wir spezifizieren wieder das System *Merge*, das wir in Beispiel 3.3 schon kennengelernt haben. Die Ein- und Ausgabekanäle des Transitionssystems *Merge* $= (I, O, A, \Upsilon, T)$ sind offensichtlich

$$I = \{i_1, i_2\} \qquad O = \{o\}$$

Da wir uns keine Daten merken müssen, und unsere Reaktion auch nicht von der Vorgeschichte abhängt, brauchen wir weder Daten- noch Kontrollzustände, und definieren ein minimales A durch Hinzufügen der Variablen für den gelesenen und ungelesenen Teil der Eingabeströme:

$$A = \{i_1^\circ, i_1^+, i_2^\circ, i_2^+\}$$

Die Startzustände beschreiben wir durch das Prädikat

$$\Upsilon \equiv i_1^\circ = \langle\rangle \wedge i_1^\circ = \langle\rangle \wedge o = \langle\rangle$$

Das System *Merge* startet also ohne initiale Ausgabe und natürlich ohne bereits konsumierte Eingaben.

Es soll nun in der Lage sein, Daten von beiden Kanälen auf den Ausgabekanal zu kopieren. Für den Kanal i_1 definieren wir dazu ein Prädikat τ_1 durch

$$\tau_1 \stackrel{df}{=} \exists\, d \bullet \quad \begin{array}{l} ft.i_1^+ = d \wedge \\ i_1^{\circ\prime} = i_1^\circ \frown \langle d\rangle \wedge \\ o' = o \frown \langle d\rangle \wedge \\ i_2^{\circ\prime} = i_2^\circ \wedge \\ i_1 \sqsubseteq i_1' \wedge i_2 \sqsubseteq i_2' \end{array}$$

Die erste Zeile stellt sicher, dass eine Nachricht d auf dem Eingabekanal i_1 vorliegt. Entsprechend der zweiten und dritten Zeile wird diese Nachricht durch diese Transition konsumiert, also in i_1° aufgenommen, und auch auf o ausgeben. Die nächsten Zeilen stellen sicher, dass keine Nachrichten von i_2 konsumiert werden, die Umgebung aber im gleichen Schritt beliebige Nachrichten an *Merge* schicken darf. Das Lesen einer Nachricht von i_2 wird durch τ_2 analog definiert.

Transition τ_1 ist genau dann schaltbereit, wenn eine Nachricht auf i_1 vorliegt. □

In den nächsten beiden Abschnitten werden wir kompaktere graphische und tabellarische Beschreibungstechniken für Transitionen angeben, bei denen auf die explizite Aufnahme von Aussagen über unveränderte Werte oder die Handlungsfreiheit der Umgebung verzichtet werden kann.

Die Attraktivität von Transitionsystemen liegt in ihrer abstrakten Formalisierung von Systemen bei gleichzeitiger Darstellung von operationellem Verhalten. Um einem System ein *Verhalten* zuordnen zu können, definieren wir die Abläufe eines Systems.

Definition 3.7 *Ablauf*

Sei ein Transitionssystem $\mathcal{S} = (I, O, A, \Upsilon, T)$ gegeben. Ein Ablauf ξ *ist eine unendliche Sequenz von Zuständen des Systems*

$$\xi = \langle\alpha, \beta, \gamma, \ldots\rangle$$

mit folgenden Eigenschaften:

- *Das System startet in einem Initialzustand, also*

  $$\xi.1 \models \Upsilon$$

- *Die Übergänge zwischen zwei in einem Ablauf aufeinanderfolgenden Zuständen sind zulässige Transitionen:*

$$\forall j \in \mathbb{N} \bullet (\xi.j, \xi.(j+1)) \in T \cup T^{\epsilon}$$

- *Sind in Zuständen mehrere Transitionen schaltbereit, so geschieht ihre Auswahl (schwach) fair, d.h. jede Transition* (α, β) *ist unendlich oft nicht schaltbereit, oder wird unendlich oft für einen Schritt verwendet:*

$$(\forall k \exists l \bullet \neg\, \xi.l \stackrel{V}{=} \alpha) \quad \vee \quad (\forall k \exists l \bullet \xi.l \stackrel{V}{=} \alpha \wedge \xi.(l+1) \stackrel{V}{=} \beta)$$

Die Menge aller Abläufe eines Transitionssystems $\mathcal{S}$ *nennen wir* $\langle\!\langle \mathcal{S} \rangle\!\rangle$*.* □

Die dritte Eigenschaft der schwachen Fairness ermöglicht es uns, Fortschrittsaussagen formal nachweisen zu können, wie wir in Abschnitt 3.8.2 sehen werden.

Ein System kann „blockieren“, wenn es in einem Zustand keine schaltbereiten Transitionen mehr gibt. Da die Umgebungstransitionen in T^{ϵ} aber immer schaltbereit sind, gibt es für jedes System immer unendliche Abläufe, auch wenn sich nach einem blockierenden Zustand die Werte der kontrollierten Variablen unter Umständen nicht mehr ändern. Schritte, in denen alle Werte unverändert bleiben (Stotterschritte), sind jederzeit möglich, da (α, α) in der Menge T^{ϵ} mit enthalten ist.

Um einem Transitionssystem ein Black-Box Verhalten zuordnen zu können, betrachten wir die Werte der gelesenen Eingabeströme und der produzierten Ausgabeströme am fiktiven „Ende“ einer Ausführung. Da Ausführungen unendlich sind, ist dies natürlich nicht direkt möglich. Da aber sowohl die Ein- als auch Ausgabeströme schrittweise verlängert werden und somit eine Kette bilden, gibt es für $v \in I \cup O$ eine kleinste obere Schranke und wir können diese als „letzten“ Wert im Ablauf definieren:

$$(\xi.\infty)(v) \stackrel{df}{=} \bigsqcup\{(\xi.k)(v) \mid k \in \mathbb{N}\}$$

Als Black-Box Verhalten eines Transitionssystems definieren wir alle Stromtupel, die im Grenzwert einer Systemausführung auf den Kanälen gelesen bzw. produziert wurden[1]:

$$\begin{aligned}[\![\mathcal{S}]\!] \stackrel{df}{=} \{(x_1, \ldots, x_n, y_1, \ldots, y_m) \mid \exists \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \\ \forall j \in \{1, \ldots, n\} \bullet (\xi.\infty)(i_j) = x_j \wedge \\ \forall j \in \{1, \ldots, m\} \bullet (\xi.\infty)(o_j) = y_j\}\end{aligned}$$

Eine ausführliche Diskussion dieser Definitionen findet sich in [10]. Für die Verhaltensrelation eines wohldefinierten Transitionssystems ist sichergestellt, dass sie linkstotal und monoton ist. Die Monotonie folgt unmittelbar aus der Forderung (d) aus Definition 3.5. Da die Umgebungstransition immer schaltbereit ist, kann auf den Eingabekanälen jeder beliebige Eingabestrom erzeugt werden.

Wählt man eine geeignete Darstellungsform für Transitionssysteme, so können darin nur wohldefinierte Systeme dargestellt werden. Zwei derartige Darstellungsformen werden wir nun präsentieren.

[1] Wir müssen hier unterscheiden zwischen den Kanalnamen i_j bzw. o_j und den Strömen x_j bzw. y_j, die auf den Kanälen transportiert werden.

3.6.1 Graphische Darstellung

Ein Transitionssystem hat eine einfache Funktionsweise: Es befindet sich in einem Zustand, und geht, ausgelöst durch Eingaben, in einen Nachfolgezustand über, wobei es Ausgaben produzieren und lokalen Variablen neue Werte zuweisen kann. Wählt man Kontrollzustände als Knoten und die Transitionen als Kanten, so lassen sich Transitionssysteme sehr intuitiv als gerichtete Graphen darstellen, wie beispielsweise in [18, 34] gezeigt.

Die Knoten stellen wir durch Ovale dar, die mit dem Namen von Kontrollzuständen beschriftet werden. Der oder die Startknoten werden mit einem schwarzen Kreis markiert. Die Beschriftung der Transitionskanten besteht aus vier Bestandteilen und hat die Form

$$\{Pre\}\ Inputs\ \triangleright\ Outputs\ \{Post\}$$

Das Eingabemuster *Inputs* besteht aus einer Liste von Ausdrücken der Gestalt

$$i?d$$

mit $i \in I$ und d als transitionslokaler Variable oder Konstante vom Typ $type(i)$. Eine Transition kann nur schalten, wenn die aktuellen Eingaben zu diesem Eingabemuster passen, also auf den entsprechenden Eingabekanälen tatsächlich Nachrichten vorliegen, und diese entweder identisch sind mit der Konstante, oder die (für eine spätere Bezugnahme) der transitionslokalen Variablen zugewiesen werden. Eine weitere Bedingung an die Schaltbereitschaft ist die Gültigkeit des Prädikates *Pre*, welches Datenvariablen und auch die transitionslokalen Variablen in $free(Pre)$ enthalten darf. In *Pre* können also weitere Vorbedingungen formuliert werden.

Wird eine Transition ausgeführt, so produziert sie Ausgaben entsprechend der Liste *Outputs* von Ausdrücken der Form

$$o!exp$$

mit o als Name eines Ausgabekanals $o \in O$ und exp als Ausdruck für den Wert, der auf diesem Kanal gesendet wird. In *Post* wird die Nachbedingung formuliert, die typischerweise den Datenvariablen neue Werte zuweist, und die daher die Variablen des Systems auch in gestrichener Form enthalten darf. *Post* muss erfüllbar sein, wenn eine Transition schaltbereit ist. Bei einfachen Zuweisungen neuer Werte an Datenvariable ist dies sichergestellt.

Aus einem Transitionsdiagramm zusammen mit der Definition der Datenvariablen mit ihrer Initialisierung und der Definition der Schnittstelle ergibt sich eindeutig die formale Darstellung des zugehörigen Transitionssystems. Die systematische Übersetzung wird in [13] beschrieben.

Beispiel 3.6 *Merge* als graphisches Transitionsdiagramm

Wir spezifizieren das System *Merge* durch ein Zustandsübergangsdiagramm in Abbildung 3.4.

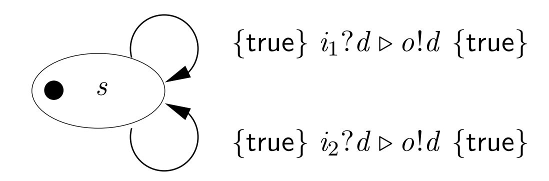


Abbildung 3.4: Zustandsübergänge von *Merge*

Wir benötigen in diesem Fall keine Datenvariablen, so dass wir auch keine Initialisierung angeben müssen. Diese wird üblicherwiese in einem Diagramm in kommentarähnlicher Form angefügt. Die Variable d ist in beiden Transitionen eine lokale Transitionsvariable. Das Ergebnis einer systematischen Übersetzung der Transitionen führt genau zu der Formalisierung, wie wir sie bereits in Beispiel 3.5 gesehen haben. Auf eine Formalisierung des Kontrollzustandes σ haben wir verzichtet, da es nur einen Zustand gibt. Beide Transitionen wären sonst um $\sigma = s$ in *Pre* und $\sigma' = s$ in *Post* ergänzt worden. □

Eine ausführliche Behandlungen von Zustandstransitionssystemen, die auch hierarchisch sein können, findet sich in [18, 33].

3.6.2 Tabellarische Darstellung

Werden Systeme sehr groß und besitzen sie sehr viele Kontrollzustände, kann die graphische Darstellung in Diagrammen sehr unübersichtlich werden. In solchen Fällen kann eine tabellarische Darstellung besser geeignet sein.

Die Transitionen werden nach dem gleichen Muster wie bei der graphischen Darstellung der Transitionssysteme notiert. Kontrollzustände müssen aber explizit aufgenommen werden als eine Datenvariable (zum Beispiel σ), die als Werte die Namen der Kontrollzustände annehmen kann. Ihre Übergänge müssen explizit angegeben werden. Die tabellarische Darstellung wird auf analoge Weise in die formale Darstellung übersetzt wie dies bereits bei der graphischen Darstellung möglich ist.

Es sind auch Mischformen beider Darstellungen gebräuchlich. Dabei werden in der graphischen Darstellung die Transitionen nur mit ihrem Namen beschriftet, und die vier Bestandteile in einer Tabelle angegeben. Damit kann die Lesbarkeit großer Diagramme erhöht werden – bei Erhalt einer intuitiven Darstellung des Kontrollflusses.

Beispiel 3.7 Wir stellen die Transitionen von *Merge* als Tabelle auf offensichtliche Weise folgendermassen dar:

Name	*Pre*	*Input*	*Output*	*Post*
τ_1	–	$i_1?d$	$o!d$	–
τ_2	–	$i_2?d$	$o!d$	–

□

In Kapitel 4 werden wir weitere Beispiele der Verwendung von Tabellen zur Beschreibung von Fehlern präsentieren.

3.7 Komposition

Komplexere Systeme werden typischerweise aus Teilkomponenten zusammengesetzt, die nach Möglichkeit unabhängig entwickelt werden können. In diesem Abschnitt wollen wir beschreiben, wie Komponenten zusammengesetzt werden können. Dazu definieren wir zunächst, unter welchen Voraussetzungen Komponenten zusammengesetzt werden dürfen und welche Schnittstelle sich für ein zusammengesetztes System ergibt. Wir geben dann sowohl für Black-Box Spezifikationen als auch für Transitionssysteme an, wie das Verhalten des Gesamtsystems aus dem Verhalten der Komponenten abgeleitet wird.

Eine für die Komposition vorteilhafte Eigenschaft unseres Systemmodells ist es, dass Systeme bzw. Komponenten nur über ihre Ein- und Ausgabekanäle mit ihrer Umgebung interagieren, und keine weiteren Seiteneffekte eine Rolle spielen. Ein komponiertes System entsteht daher auf einfache Weise aus den Einzelkomponenten durch die Verschaltung der Kanäle. Eine Verschaltung muss angeben, welche Komponenten welche Ausgaben einer anderen Komponente als Eingabe erhält.

Da alle Komponenten auf Kanäle zugreifen, die mit konkreten Namen referenziert werden, ergibt sich eine sehr einfache Technik zur Beschreibung der Komposition: Kanäle mit gleichem Namen werden identifiziert. Ein Konflikt kann nur auftreten, wenn zwei zu komponierende Systeme gemeinsame Ausgabekanäle haben, da dann beide auf dem gleichen Kanal Nachrichten senden würden. Diesen Fall schließen wir mit Hilfe der folgenden Definition aus:

Definition 3.8 *Komposition*

Zwei Komponenten S und T mit den Schnittstellen (I_S, O_S) und (I_T, O_T) heißen kompatibel*, wenn sie keine gemeinsamen Ausgabekanäle haben, also*

$$O_S \cap O_T = \varnothing$$

Das aus S und T zusammengesetzte System wird bezeichnet durch

$$S \otimes T$$

und besitzt als Schnittstelle

$$\begin{aligned} I_{S\otimes T} &\stackrel{df}{=} (I_S \cup I_T) \setminus (O_S \cup O_T) \\ O_{S\otimes T} &\stackrel{df}{=} O_S \cup O_T \end{aligned}$$

□

In Abbildung 3.5 ist die entstehende Schnittstelle veranschaulicht. Die Eingabeschnittstelle des Gesamtsystems besteht also aus den Eingabekanälen der beiden Komponenten ohne diejenigen Kanäle, die von den Ausgaben der jeweils anderen Komponente schon mit Nachrichten versorgt werden. Diese Kanäle bleiben aber in der Ausgabe sichtbar, so dass sich die Ausgabeschnittstelle als Vereinigung aller Ausgabekanäle ergibt. *Interne* Kanäle entstehen somit nicht. Wir haben diese Wahl getroffen, da sie eine Vereinfachung

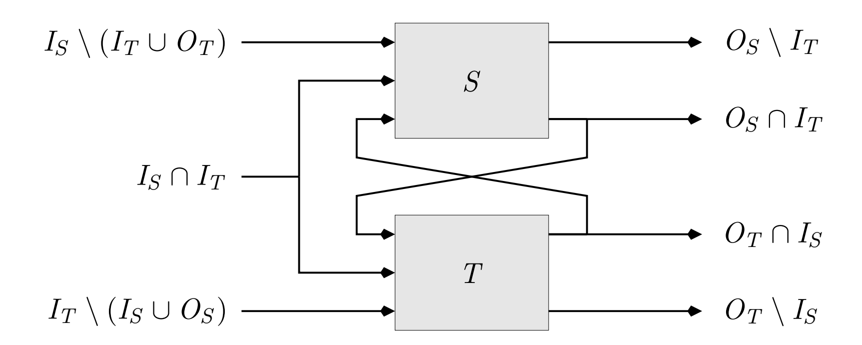


Abbildung 3.5: Komposition von Komponenten

für die Verifikation mit sich bringt, indem auf alle Informationen offen zugegriffen werden kann.

Die Komposition ist nur für zwei Komponenten definiert. Durch iterative Anwendung können aber beliebig viele Komponenten zu einem System verbunden werden. Die Reihenfolge der Komposition spielt dabei keine Rolle, denn sind die Komponenten paarweise kompatibel, so sind die resultierenden Schnittstellen (und das Verhalten) von $S \otimes (T \otimes U)$ und $(S \otimes T) \otimes U$ identisch. Sind die Komponenten jedoch nicht paarweise kompatibel, kann dennoch nach einer geeigneten Umbenennung der Kanäle eine Komposition möglich sein.

Eine Umbenennung kann aus verschiedenen Gründen notwendig werden: Bei der Spezifikation eines Systems können die Kanalnamen willkürlich gewählt werden. Möchte man Systeme komponieren, bei deren Entwurf nicht auf eine geeignete Namensgebung geachtet wurde, so können die System unnötigerweise nicht kompatibel sein, wenn die Ausgabekanäle den gleichen Namen tragen. Die Kanalnamen können vollkommen auch disjunkt sein, so dass die Komposition nur eine parallele und vollkommen interaktionsfreie „Komposition“ der Systeme ergibt.

Eine Umbenennung erfordert die Zuordnung von alten auf neue Kanalnamen. Jedem alten Kanalnamen wird genau ein neuer Kanalname zugeordnet. Eine Umbenennung können wir also definieren durch eine bijektive Funktion zwischen Kanalnamen. Bei einer Komposition können nur Kanäle identifiziert werden, die den gleichen Nachrichtentyp aufweisen. Daher muß die Umbenennung den Typ der Kanäle erhalten. Können Kanalnamen aus zwei Mengen typerhaltend umbenannt werden, nennen wir die Mengen der Kanalnamen *vereinbar*:

Definition 3.9 *Vereinbarkeit von Schnittstellen, Umbenennung*

Seien zwei Mengen X und Y von Kanalnamen gegeben. Sie heißen vereinbar*, im Zeichen*

$$X \Leftrightarrow Y$$

wenn es eine Bijektion $f : X \to Y$ gibt, so dass

$$\forall\, c \in X \bullet type(c) = type(f(c))$$

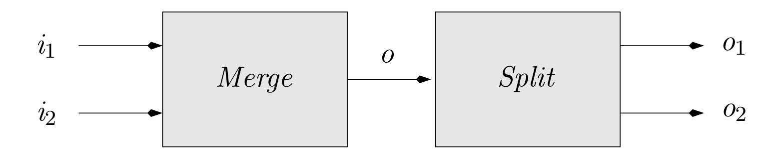


Abbildung 3.6: Multiplexer

Sie heißen zusätzlich eindeutig vereinbar*, wenn es genau eine solche Bijektion gibt. Die Bijektion f nennen wir* Umbenennung. □

Wir werden diese Definition in Kapitel 4.2.4 verwenden. Im folgenden Beispiel ist eine Umbenennung nicht notwendig, da die Komponenten von vornherein passende Kanalnamen aufweisen.

Beispiel 3.8 Spezifikation eines Multiplexers

Wir definieren zunächst eine weitere Komponente *Split*, die wir mit *Merge* komponieren wollen. Diese Komponente empfängt den Datenstrom, den *Merge* erzeugt, und trennt die Daten wieder in zwei Datenströme auf. *Split* besitzt also $\{o\}$ als Eingabekanal. Wir definieren die Ausgabekanäle zu $\{o_1, o_2\}$ mit den Typen $type(o_j) = D_j$. Damit ist *Split* also offensichtlich kompatibel zu *Merge*. Das komponierte System ist in Abbildung 3.6 dargestellt. Als Eingabeschnittstelle des Systems

$$Multiplex \stackrel{df}{=} Merge \otimes Split$$

ergeben sich $\{i_1, i_2\}$ als Eingabekanäle und $\{o_1, o_2, o\}$ als Ausgabekanäle. □

In den folgenden beiden Abschnitten definieren wir für unsere beiden Beschreibungstechniken, wie sich das Verhalten eines komponierten Systems aus dem Verhalten der Einzelkomponenten ergibt.

Im Rahmen eines Entwicklungsprozesses ist es empfehlenswert, die Komposition auf der Ebene der Black-Box Spezifikationen durchzuführen. Wir werden dies in Abschnitt 3.10 diskutieren. In manchen Fällen lassen sich allerdings für Systeme nur komplizierte und unhandliche explizite Black-Box Spezifikationen finden, oder es gehen durch die damit verbundene Abstraktion Informationen verloren, die eine Verifikation erschweren können. Für derartige Situationen ist ein Formalismus zur Komposition von Zustandsmaschinen hilfreich.

In Kapitel 4.5.2 werden wir auf der Basis der hier vorgestellten Komposition die Auswirkungen ableiten, die Fehler in Komponenten auf die Gesamtsysteme haben, die sie konstituieren.

3.7.1 Komposition von Black-Box Spezifikationen

Die Black-Box Spezifikation eines komponierten Systems, deren Komponenten selbst durch Black-Box Spezifikationen beschrieben sind, ist sehr einfach definiert als Konjunktion:

Definition 3.10 *Black-Box Komposition*

Seien S^Φ und T^Φ Black-Box Spezifikationen zweier kompatibler Komponenten S und T. Das Verhalten von $S \otimes T$ ist beschrieben durch

$$S^\Phi \wedge T^\Phi$$

□

Ein Stromtupel von Ein- und Ausgabeströmen beschreibt ein mögliches Verhalten eines komponierten Systems, wenn diese Stromtupel auch einem möglichen Verhalten der Einzelkomponenten entsprechen.

Beispiel 3.9 Black-Box Spezifikation des Multiplexers

Die Komponente *Split* können wir durch folgende Black-Box Spezifikation definieren:

$$Split^\Phi \stackrel{df}{=} o_1 = D_1 \circledS o \quad \wedge \quad o_2 = D_2 \circledS o$$

Für $j = 1, 2$ erscheinen auf dem Kanal o_j alle Nachrichten von o, die in der Menge D_j liegen, und zwar auch in der gleichen Reihenfolge.

Das Verhalten des zusammengesetzten Systems *Multiplex* ergibt sich damit direkt zu der Konjunktion aus $Merge^\Phi$ und $Split^\Phi$:

$$\begin{aligned} Multiplex^\Phi \equiv \; & i_1 = D_1 \circledS o \quad \wedge \quad i_2 = D_2 \circledS o \quad \wedge \\ & o_1 = D_1 \circledS o \quad \wedge \quad o_2 = D_2 \circledS o \end{aligned}$$

Aus dieser Verhaltensbeschreibung folgt $i_1 = o_1 \wedge i_2 = o_2$. Dies beschreibt damit ein Verhalten, wie man es von einem Multiplexer erwartet. □

3.7.2 Komposition von Zustandsmaschinen

Zwei Zustandsmaschinen können zu einer Zustandsmaschine komponiert werden, die genau die Transitionen ausführen kann, die auch die beiden Komponenten durchführen könnten: Das zusammengesetzte System kombiniert dazu eine Transition der einen Teilmaschine mit einer Umgebungstransition der anderen Maschine, und führt sie gemeinsam aus. Damit keine Konflikte durch Überlappungen von internen Variablen entstehen, müssen (neben der Kompatibilität der Schnittstellen) weitere Forderungen gestellt werden:

Definition 3.11 *Komposition von Zustandsmaschinen*

Es seien zwei kompatible Zustandsmaschinen gegeben mit $\mathcal{S}_1 = (I_1, O_1, A_1, \Upsilon_1, T_1)$ und $\mathcal{S}_2 = (I_2, O_2, A_2, \Upsilon_2, T_2)$ für die (durch eventuelle Umbenennung) sichergestellt sei, dass

$$\begin{aligned} & (O_1 \cup A_1) \cap (O_2 \cup A_2) = \varnothing \wedge \\ & A_1 \cap I_2 = \varnothing \wedge A_2 \cap I_1 = \varnothing \end{aligned}$$

Das Verhalten des komponierten Systems $\mathcal{S}_1 \otimes \mathcal{S}_2$ ist definiert durch die Zustandsmaschine $\mathcal{S} = (I, O, A, \Upsilon, T)$ mit I und O definiert wie in Definition 3.8 und

$$\begin{aligned} A &= A_1 \cup A_2 \\ \Upsilon &= \Upsilon_1 \wedge \Upsilon_2 \\ T &= (T_1 \cap T_2^\epsilon) \cup (T_2 \cap T_1^\epsilon) \end{aligned}$$

□

Das komponierte System besitzt alle Variablen der Teilsysteme als lokale Variable. Gemäß Voraussetzung gibt es hier keine Konflikte. Das System startet in einem Zustand, der die Initialbedingungen beider Komponenten erfüllt. Die Schnittmengenbildung im Ausdruck für die Transitionsmenge T stellt sicher, dass jedes Belegungspaar (α, β) gleichzeitig eine Transition der einen Komponente und eine Umgebungstransition der jeweils anderen Komponente darstellt. Die Zusicherung, dass eine Transition immer die Umgebungstransition von jeweils einer Komponente ist stellt sicher, dass sich darin die lokalen Variablen nicht spontan ändern können. Die Transition einer Teilkomponente hat (aufgrund der vorausgesetzten Disjunktheit der kontrollierten Variablen) außer einer Verlängerung der Eingabeströme keine Wirkung auf die jeweils andere Komponente, also gibt es für sie immer eine passende Umgebungstransition.

Falls beide Transitionssysteme die Eigenschaften aus Definition 3.5 erfüllen, erfüllt auch das komponierte System wieder diese Eigenschaften (gezeigt in [10]), ist also wieder ein wohlgeformtes Transitionssystem.

3.8 Verifikation von Eigenschaften

Mit den bisher dargestellten Techniken sind wir in der Lage, Systeme sowohl durch Black-Box Spezifikationen als auch durch Transitionssysteme darzustellen. Black-Box Spezifikationen charakterisieren ein Systemverhalten durch eine Formel Φ, die damit die Eigenschaften des Ein-/Ausgabeverhaltens unmittelbar beschreibt. Ein Transitionssystem gibt dagegen ein Verhalten operationell an und enthält keine unmittelbaren Aussagen über die Eigenschaften des Systemverhaltens. An der Transitionsmenge der möglichen Einzelschritte ist das Verhalten, das daraus entstehen könnte, unter Umständen schwer erkennbar. Dennoch ist man an der Verifikation von Eigenschaften der Abläufe von Transitionssystemen interessiert, da sich damit ein Zusammenhang zwischen den beiden Beschreibungstechniken herstellen läßt.

Wir werden im folgenden mit den Invarianten und den Fortschrittseigenschaften zwei Klassen von Eigenschaften der Transitionssysteme vorstellen, die mit den bekannten *Sicherheitseigenschaften* beziehungsweise *Lebendigkeitseigenschaften* (diese werden vorgestellt in [1]; ein Literaturüberblick findet sich in [40]) eng verwandt sind. Daraufhin werden wir die Technik der Verifikationsdiagramme präsentieren, die sich eignet, formale Beweise auf abstraktem Niveau intuitiv darzustellen. Eine ausführliche Behandlung dieser Themen findet sich in [10, 13].

3.8.1 Eigenschaften von Transitionssystemen

Die Eigenschaft P eines Transitionssystems beschreiben wir durch ein Prädikat P. Weist ein System $\mathcal{S}$ ein Prädikat P auf, so notieren wir dies als

$$\mathcal{S} \Vdash P$$

Wir sind hier nicht an statischen Eigenschaften eines Systems interessiert (wie beispielsweise die Zahl der Transitionen, den Grad von Nichtdeterminismus oder die Anzahl der

Kanäle), sondern an Aussagen über den dynamischen Anteil, also die *Abläufe* eines Systems.

Wir beschränken uns auf Eigenschaften, die sich durch *Invarianten* und durch *Fortschrittsaussagen* ausdrücken lassen. Dazu werden Zustandsprädikate verwendet, um Aussagen über die Menge der Ausführungen von Systemen zu machen. Sei $\mathcal{S}$ im folgenden gegeben durch $\mathcal{S} = (I, O, A, \Upsilon, T)$.

Definition 3.12 *Invarianten*

Ein Zustandsprädikat Φ eines Transitionssystems $\mathcal{S}$ heißt Invariante *von $\mathcal{S}$, notiert als*

$$\mathcal{S} \Vdash \textbf{inv}\ \Phi$$

wenn Φ in allen Abläufen in allen erreichbaren Zuständen wahr ist:

$$\forall \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \forall k \bullet \xi.k \models \Phi$$

□

Die Gültigkeit einer Invariante wird typischerweise gezeigt, indem man nachweist, dass die Invariante im Initialzustand gültig ist, und bei jeder möglichen Transition erhalten bleibt, wie dies in der folgenden Beweisregel formalisiert ist:

$$\begin{array}{|l}
\Upsilon \Rightarrow \Phi \\
\Phi \wedge \tau \Rightarrow \Phi' \quad \text{für alle} \quad \tau \in T \\
\Phi \wedge \tau^{\varepsilon} \Rightarrow \Phi' \quad \text{für alle} \quad \tau^{\varepsilon} \in T^{\epsilon} \\
\hline
\mathcal{S} \Vdash \textbf{inv}\ \Phi
\end{array}$$

Nicht jede Invariante läßt sich auf diese Weise zeigen, da eine Invariante zu schwach sein kann und damit unerreichbare Zustände umfaßt, für die die Beweisregel unnötige Beweisverpflichtungen erzeugt. Die Invariante ist in einem solchen Fall geeignet zu verstärken. Die stärkste Invariante eines Systems charakterisiert exakt die erreichbaren Zustände eines Systems.

Dieses Beweisprinzip wird auch in einem Beweisdiagramm für Invarianten verwendet, das im nächsten Abschnitt vorgestellt wird. In [10] ist eine Reihe weiterer Beweisregeln zum Nachweis von Invarianten angegeben.

Invarianten sind leicht erfüllbar von einem System, das keinerlei Aktivität zeigt, beispielsweise von einem trivialen System ohne Transitionen oder einem blockierten System. Daher fordert man üblicherweise eine gewisse minimale Aktivität. Wir formulieren diese mit Hilfe sogenannter Fortschrittseigenschaften, die sich wieder auf Zustandsprädikate abstützen:

Definition 3.13 *Fortschrittseigenschaften*

Ein Zustand Φ zieht in einem System $\mathcal{S}$ einen Zustand Ψ nach sich, notiert als

$$\mathcal{S} \Vdash \Phi \mapsto \Psi$$

wenn in all seinen Abläufen nach Erreichen eines Zustandes, in dem Φ gilt, auch sofort Ψ gilt oder im weiteren Ablauf ein Zustand folgt, in dem Ψ gilt:

$$\forall \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \forall k \bullet (\xi.k \models \Phi) \Rightarrow (\exists l \geq k \bullet \xi.l \models \Psi)$$

□

In [10] finden sich wieder einige Beweisregeln zum Nachweis einer Fortschrittseigenschaft. Im Rahmen unseres Systemmodells ist man häufig an der folgenden speziellen Form interessiert, die eine Verlängerung der Ausgabe ausdrückt, die ein System leisten soll:

$$\#o = k \wedge k < \ell \mapsto \#o > k$$

Dabei beschreibt ℓ einen ganzzahligen Ausdruck mit freien Variablen aus $I \cup O$. Erfüllt ein System diese Eigenschaft und hat in einem Zustand die Länge des Ausgabestromes den Wert ℓ noch nicht erreicht, so wird der Ausgabestrom im weiteren Systemablauf noch verlängert. Es ist in diesem System also sichergestellt, dass die Länge des Ausgabestroms schließlich die Länge ℓ erreichen wird. Üblicherweise hängt ℓ von den Eingaben ab, hat also Elemente aus I als freie Variablen.

Zum Nachweis dieser Aussage eignet sich folgende Regel, die die Existenz einer zielführenden Transition fordert, die schaltbereit ist und die Ausgabe tatsächlich verlängert. Aufgrund der Fairness in unserem Systemmodell wird diese Transition also tatsächlich einmal ausgeführt und damit die Ausgabe wie gefordert verlängert. Der Beweis der Regel findet sich wieder in [10].

Für eine Transition $\tau \in T$:

$$\#o = k \wedge k < \ell \Rightarrow \mathsf{En}(\tau)$$

und

$$\#o = k \wedge k < \ell \wedge \tau \Rightarrow \#o' > k$$

$$\mathcal{S} \Vdash \#o = k \wedge k < \ell \leadsto \#o > k$$

In den Anwendungsbeispielen in [10, 11, 12, 13] haben sich Aussagen zur Invarianz und zur Ausgabeverlängerung als ausreichend erwiesen, um die interessanten Eigenschaften eines Transitionssystems auszudrücken. Während mit Invarianten die Korrektheit der ausgegebenen Nachrichten formalisiert wird, gibt die Aussage über die Ausgabenverlängerung an, dass ausreichend viele Ausgaben gemacht werden.

Beispiel 3.10 Eigenschaften von *Merge*

Wir greifen wieder das System *Merge* auf. Als Invariante formulieren wir die Aussage, dass in jedem Zustand in jedem Ablauf des Systems genau so viele Daten ausgegeben sind, wie auf den beiden Eingabekanälen bislang konsumiert wurden:

$$\mathit{Merge} \Vdash \textbf{inv}\ \#o = \#i_1^\circ + \#i_2^\circ$$

Im nächsten Abschnitt werden wir die Gültigkeit dieser Invariante beweisen.

Als Beispiel für eine Fortschrittseigenschaft geben wir folgende Behauptung an: Wird in einem Ablauf ein Zustand erreicht, in dem die Länge des Ausgabestromes kürzer ist als die Summe der Längen der Eingabeströme, so wird im weiteren Ablauf die Ausgabe noch weiter verlängert.

$$\mathit{Merge} \Vdash \#o = k \wedge k < \#i_1 + \#i_2 \quad \mapsto \quad \#o > k$$

Auch diese Aussage werden wir im folgenden mit Hilfe von Verifikationsdiagrammen beweisen.
□

Da in komponierten Systemen die Teilkomponenten nur über asynchrone und damit nicht-blockierenden Nachrichtenaustausch kommunizieren und interagieren, sonst jedoch entkoppelt bleiben, weist unser Systemmodell *Kompositionalität* auf, d.h. die Eigenschaften eines Systems bleiben im Kontext eines weiteren Systems erhalten. Es gilt also (für $j \in \{1,2\}$)

$$\frac{\mathcal{S}_j \Vdash \textbf{inv } \Phi}{\mathcal{S}_1 \otimes \mathcal{S}_2 \Vdash \textbf{inv } \Phi} \qquad \frac{\mathcal{S}_j \Vdash \Phi \mapsto \Psi}{\mathcal{S}_1 \otimes \mathcal{S}_2 \Vdash \Phi \mapsto \Psi}$$

3.8.2 Verifikationsdiagramme

Ein formaler Beweis einer Eigenschaft eines Transitionssystems stellt sich klassischerweise als eine lineare Folge von Aussagen dar, die ausgehend von Axiomen oder Tautologien unter Verwendung von Beweisregeln über Zwischenresultate hinweg die zu beweisende Aussage herleiten. Eine derartige Darstellung folgt nicht der Intuition der operativen Funktionsweise, die man mit einem Transitionssystem verbindet. Ein Verifikationsdiagramm ist geeignet, einen Beweis zu veranschaulichen. Es stellt an sich keinen Beweis dar, aber impliziert eine Menge von einzelnen Beweisverpflichtungen. Können diese nachgewiesen werden, so ist ein Diagramm *gültig* und repräsentiert eine Gesamtaussage. Verifikationsdiagramme wurden in [14, 50] eingeführt und in [11] für unser Systemmodell spezialisiert.

Wir unterscheiden Diagramme zum Nachweis von Invarianten und Diagramme zum Nachweis von Fortschrittseigenschaften. Beide haben aber die folgenden Charakteristika gemeinsam:

Ein Verifikationsdiagramm (zu einem Transitionssystem $\mathcal{S} = (I, O, A, \Upsilon, T)$) besteht aus einem gerichteten, zusammenhängenden Graphen mit einer endlichen Menge von Knoten, die mit Prädikaten $\Phi_0, \ldots, \Phi_n$ beschriftet sind. Die Prädikate sind Zustandsprädikate des Transitionssystems, also mit $\mathit{free}(\Phi_j) \subseteq I \cup O \cup A$. Die Kanten des Diagramms sind mit (Mengen von) Transitionen beschriftet.

Invariantendiagramme

Ein Diagramm zum Nachweis einer Invariante erfüllt zusätzlich zu den eben genannten die folgenden Eigenschaften:

- Bei Start des Systems $\mathcal{S}$ ist mindestens eines der Prädikate erfüllt. Die erfüllten Prädikate werden (mit einem schwarzen Punkt links in einem Knoten) als Startzustände markiert.

  $$\Upsilon \Rightarrow \Phi_0 \vee \ldots \vee \Phi_n$$

- Für jeden Knoten Φ_j und jede Transition $\tau \in T$ gilt genau einer der beiden folgenden Fälle:
  - Vom Knoten Φ_j geht *keine* mit τ beschriftete Kante aus, und es gilt

    $$\Phi_j \wedge \tau \Rightarrow \Phi'_j$$

    Dieser Fall liegt auch dann vor, wenn τ im Knoten Φ_j nicht schaltbereit ist. Die geforderte Implikation gilt trivialerweise, da $\Phi_j \wedge \tau$ dann den Wahrheitswert false hat.
  - Vom Knoten Φ_j gehen mit τ beschriftete Kanten zu den Knoten $\Phi_{i_1}, \ldots, \Phi_{i_k}$, und es gilt

    $$\Phi_j \wedge \tau \Rightarrow \Phi'_{i_1} \vee \ldots \vee \Phi'_{i_k}$$

- Die Umgebungstransitionen erhalten jedes Prädikat, es gilt also für alle $\tau^\varepsilon \in T^\epsilon$

  $$\Phi_j \wedge \tau^\varepsilon \Rightarrow \Phi'_j$$

Erfüllt ein Diagramm alle diese Eigenschaften, so repräsentiert es einen Beweis für die Aussage

$$\mathcal{S} \Vdash \mathbf{inv}\ \Phi_0 \vee \ldots \vee \Phi_n$$

Die Disjunktion gilt aufgrund der ersten Forderung initial und bleibt wegen der anderen Forderungen für jede Transition erhalten. Ein Beweis findet sich in [10]. Wir illustrieren die Invariantendiagramme mit Hilfe des folgenden Beispiels.

Beispiel 3.11 Wir beweisen mit Hilfe des Verifikationsdiagrammes in Abbildung 3.7, dass

$$\#o = \#i_1^\circ + \#i_2^\circ$$

eine Invariante von *Merge* ist. In allen vier Knoten gilt die geforderte Gleichung, so dass die Behauptung eine Konsequenz der Disjunktion der vier Zustandsprädikate ist.

Initial sind keine Eingaben gelesen, und noch keine Ausgaben ausgegeben, so dass Υ das Prädikat im Startknoten impliziert. Von den 12 Beweisverpflichtungen, resultierend aus 4 Knoten mal 3 Transitionen (inklusive der Umgebungstransition) greifen wir exemplarisch eine heraus:

$$\#o = \#i_1^\circ \wedge \#i_2^\circ = 0 \wedge \tau_2 \quad \Rightarrow \quad \#o' = \#i_1^{\circ\prime} + \#i_2^{\circ\prime}$$

Dies gilt offensichtlich, da aus der Definition von τ_2 folgt, dass

$$\exists\, d \bullet i_1^{\circ\prime} = i_1^\circ \wedge i_2^{\circ\prime} = i_2^\circ \frown \langle d \rangle \wedge o' = o \frown \langle d \rangle$$

Auf beiden Seiten der Gleichung der Invarianten erhöht sich der Wert um 1. Damit bleibt die Invariante also erhalten.

Das angegebene Beweisdiagramm ist nicht das einzige Diagramm, das die Invariante zeigt. Wir haben hier nicht das einfachste Diagramm gewählt, um die Anwendung der Verifikationsdiagramme besser veranschaulichen zu können. □

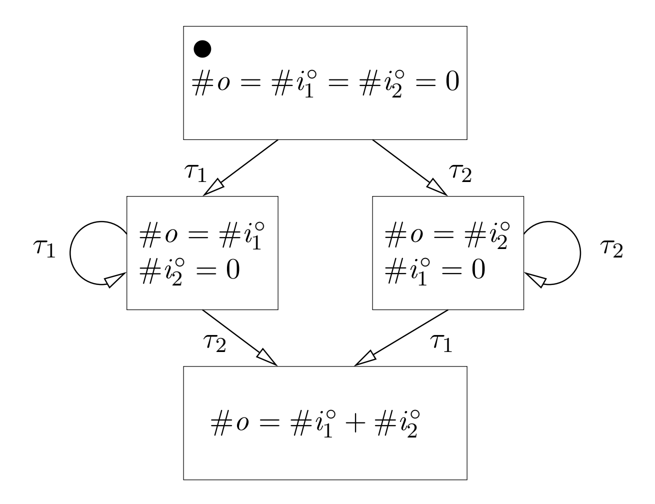


Abbildung 3.7: Invariantendiagramm für *Merge*

Fortschrittsdiagramme

Der Nachweis der Eigenschaft $\Phi \mapsto \Psi$ für alle Abläufe eines Systems kann mit einem Fortschrittsdiagramm erfolgen, das die folgenden Eigenschaften erfüllt. Die Knoten werden zusätzlich mit einer *Bewertung* δ_i versehen, die einem Systemzustand einen Wert aus einer wohlfundierten Ordnung $(W, <)$ zuordnet, also $\delta_i : \alpha \rightarrow W$.

- Das Prädikat Φ impliziert mindestens eines der Prädikate im Diagramm, also

  $$\Phi \Rightarrow \Phi_0 \vee \ldots \vee \Phi_n$$

- Jeder Knoten $\Phi_1, \ldots, \Phi_n$ hat mindestens eine ausgehende Kante. Von den ausgehenden Kanten $\tau_{i_1}, \ldots, \tau_{i_k}$ eines Knotens Φ_j ist mindestens eine Transition schaltbereit:

  $$\Phi_j \Rightarrow \mathsf{En}(\tau_{i_1}) \vee \ldots \vee \mathsf{En}(\tau_{i_k})$$

- Für jeden Knoten Φ_j mit $j \in \{1, \ldots, n\}$ und jede Transition $\tau \in T$ gilt genau einer der beiden folgenden Fälle:

  – Vom Knoten Φ_j geht *keine* mit τ beschriftete Kante aus, und es gilt

    $$\Phi_j \wedge \delta_j = u \wedge \tau \quad \Rightarrow \quad \Phi'_j \wedge \delta_j \leq u$$

  – Vom Knoten Φ_j geht eine Kante zu einem Knoten Φ_k. Für alle derartigen Knoten Φ_k gilt

    $$\Phi_j \wedge \delta_j = u \wedge \tau \quad \Rightarrow \quad \Phi'_k \wedge \delta_k < u$$

- In dem durch Φ_0 repräsentierten Zustand ist das Ziel Ψ erreicht, d.h.

  $$\Phi_0 \Rightarrow \Psi$$

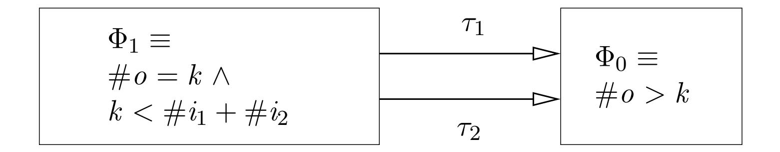


Abbildung 3.8: Fortschrittsdiagramm für *Merge*

Befindet sich das System in einem Zustand, in dem Φ gilt, so befindet es sich auch in einem durch $\Phi_0, \ldots, \Phi_n$ charakterisierten Zustand. Da immer eine Transition schaltbereit ist, wird aufgrund der Fairness jeder Zustand auch wieder verlassen. Beim Wechsel in einen anderen Knoten wird dabei der Wert der Bewertungsfunktion reduziert. Da der Wert nicht beliebig klein werden kann, muss schließlich im weiteren Systemablauf der Knoten Φ_0 erreicht werden, der das Zustandsprädikat Ψ erfüllt. Können also alle geforderten Eigenschaften eines Fortschrittsdiagramms nachgewiesen werden, repräsentiert es einen Nachweis für die Aussage

$$\mathcal{S} \Vdash \Phi \mapsto \Psi$$

Diese Form der Fortschrittsdiagramme stellt eine Verallgemeinerung der Diagramme in [10] dar, da Zyklen im Graph hier zugelassen werden. Eine Einbettung wird mit der trivialen Bewertung $\delta_i = i$ erreicht.

Beispiel 3.12 Wir wollen mit einem Diagramm beweisen, dass

$$\mathit{Merge} \Vdash \#o = k \wedge k < \#i_1 + \#i_2 \quad \mapsto \quad \#o > k$$

gilt, d.h. die Ausgabe wird verlängert, solange sie nicht genauso lang ist wie die Eingabeströme an beiden Eingabekanälen. Es werden also tatsächlich alle Eingaben wieder ausgegeben.

Das einfache Beweisdiagramm mit nur zwei Knoten Φ_1 und Φ_0 findet sich in Abbildung 3.8. Als Bewertung können wir die Konstanten $\delta_1 = 1$ und $\delta_0 = 0$ wählen. Fast alle Beweisverpflichtungen sind offensichtlich; nur die Schaltbereitschaft einer der beiden Transitionen in Zustand Φ_1 muss nachgewiesen werden.

Dazu müssen wir die in Beispiel 3.11 gezeigte Invariante $\#o = \#i_1^\circ + \#i_2^\circ$ verwenden, und können folgern, dass

$$\#i_1^\circ + \#i_2^\circ = \#o = k < \#i_1 + \#i_2 = (\#i_1^\circ + \#i_1^+) + (\#i_2^\circ + \#i_2^+)$$

im Knoten Φ_1 gültig ist, woraus sich $\#i_1^+ + \#i_2^+ > 0$ ergibt. Damit wissen wir, dass auf einem der beiden Kanäle noch ungelesene Nachrichten vorhanden sind, woraus wir $\mathsf{En}(\tau_1) \vee \mathsf{En}(\tau_2)$ schließen können. Das System *Merge* wird also noch Ausgaben produzieren. □

Wir können auch hierarchische Diagramme zulassen, in denen Knoten einander umfassen können. Dies kann dann nützlich sein und zur Übersichtlichkeit beitragen, wenn die Knoten gemeinsame Konjunktionsglieder haben, die somit nur noch einmal notiert werden müssen. Hierarchische Diagramme können in flache Diagramme übersetzt werden, die die gleiche Mächtigkeit aufweisen. Wir verzichten daher auf eine genauere Darstellung und verweisen auf [13].

Aus einem Diagramm könnten mit Hilfe geeigneter Werkzeuge die darin enthaltenen Beweisverpflichtungen generiert werden und sogar (automatisch oder interaktiv) bewiesen werden. In [13] wird dieser Ansatz vorgestellt.

In Kapitel 5.7 werden wir die Beweisdiagramme wieder aufgreifen und diskutieren, wie sie bei der Modifikation von Systemen entsprechend angepaßt werden können mit dem Ziel, möglichst wenig von dem bereits geleisteten Beweisaufwand zu verlieren.

3.8.3 Black-Box Eigenschaften von Transitionssystemen

Die Eigenschaften von Zustandsmaschinen aus Abschnitt 3.8.1, die mit den Diagrammen aus Abschnitt 3.8.2 bewiesen werden können, machen Aussagen über die Zustände des Transitionssystems. Eine Black-Box Eigenschaft dagegen spricht nur über die nach außen sichtbaren Ein- und Ausgabeströme. Der Übergang zwischen beiden Sichten kann mit zwei Regeln hergestellt werden, die in [10] hergeleitet und ausführlicher diskutiert werden. Sie lauten

$$\frac{\mathcal{S} \Vdash \textbf{inv } \Phi \qquad \textbf{adm } \Phi}{[\![\mathcal{S}]\!] \Rightarrow \Phi} \qquad\qquad \frac{\mathcal{S} \Vdash \#o = k \wedge k < \ell \mapsto \#o > k}{[\![\mathcal{S}]\!] \Rightarrow \#o \geq \ell}$$

wobei Φ und auch ℓ nur Variable aus $I \cup O$ enthalten dürfen. Das Prädikat **adm** (für *admissible*) stellt sicher, dass die Eigenschaft Φ auch für unendliche Ströme gültig ist, wenn sie für alle endlichen Approximationen erfüllt ist. Eine Invariante gilt dann auch für die unendlichen Ein-/Ausgabeströme, die im Grenzwert erreicht werden. Die Fortschrittseigenschaft führt zu einer iterierten Verlängerung einer Ausgabe, solange der Wert ℓ noch nicht erreicht ist. Im Falle eines unendlichen Ablaufs wird dieser Grenzwert letztendlich erreicht oder sogar übertroffen.

Beispiel 3.13 Black-Box Eigenschaften von *Merge*

Wir verwenden die beiden Regeln, um eine Black-Box Eigenschaft von *Merge* herzuleiten. In Beispiel 3.11 haben wir bewiesen, dass $\#o = \#i_1^\circ + \#i_2^\circ$ eine Invariante von *Merge* ist. Da auch $i^\circ \sqsubseteq i$ und damit $\#i^\circ \leq \#i$ für alle Eingabeströme i eine Invariante in jedem System ist, können wir

$$\mathit{Merge} \Vdash \textbf{inv } \#o \leq \#i_1 + \#i_2$$

folgern. Dieses Prädikat ist gemäß den Kriterien aus [10] *admissible*, so dass wir unter Anwendung obiger Regel

$$[\![\mathit{Merge}]\!] \Rightarrow \#o \leq \#i_1 + \#i_2$$

ableiten. Unter Verwendung der Regel für Fortschritteigenschaften können wir aus dem Resultat in Beispiel 3.12 folgern, dass

$$[\![\mathit{Merge}]\!] \Rightarrow \#o \geq \#i_1 + \#i_2$$

Kombiniert man beide Resultate, erhalten wir mit

$$[\![\mathit{Merge}]\!] \Rightarrow \#o = \#i_1 + \#i_2$$

die erwartete Black-Box Eigenschaft von *Merge*. □

3.9 Verfeinerungen

Die Entwicklung eines komplexen Systems kann nicht in einem einzigen Schritt erfolgen. Eine Systementwicklung muss mehrere Phasen durchlaufen, die sich von der Erfassung von Systemanforderungen bis zu einer konkreten Implementierung und Integration von Systemkomponenten erstrecken. Dabei wird ein System typischerweise zunächst relativ abstrakt beschrieben, um eine grobe Strukturierung des Systems zu ermöglichen. Im weiteren Entwicklungsprozeß werden durch Entwurfsentscheidungen mehr und mehr Details ergänzt, die alle Aspekte eines Systems betreffen können. Beispiele für Entwurfsentscheidungen sind die Wahl einer Systemarchitektur, die Wahl eines Datenmodells und die Wahl einer Systemplattform. Derartige Entscheidung haben Einfluß aufeinander und müssen daher sehr sorgfältig gewählt werden.

Die verschiedenen in einem Entwicklungsprozeß entstehenden Systemversionen sollen natürlich nicht unabhängig voneinander sein, sondern sind idealerweise durch sogenannte *Verfeinerungen* verbunden. Liegt eine Verfeinerung zwischen zwei Varianten eines Systems vor, so beschreibt die verfeinerte Variante *im wesentlichen* das gleiche System wie die abstrakte Sicht, darf aber zusätzliche Eigenschaften und Konkretisierungen aufweisen. Es ist also formal zu definieren, wann ein System eine Verfeinerung (oder *Implementierung*) eines anderes Systems darstellt.

Wir wollen dazu in diesem Abschnitt den Verfeinerungsbegriff im Rahmen unseres Systemmodells vorstellen. Dazu beschreiben wir kurz die beiden Verfeinerungsrelationen der *Verhaltens-* und der *Schnittstellenverfeinerung*. Eine deutlich ausführlichere Einführung in diese Konzepte findet sich in [21].

3.9.1 Verhaltensverfeinerung

Die Verhaltensverfeinerung erlaubt es, ein abstraktes System mit einem konkreten System in Beziehung zu setzen, das die gleiche Schnittstelle hat, aber zusätzliche Forderungen an das Verhalten aufweisen kann.

Definition 3.14 *Verhaltensverfeinerung*
Seien zwei Systeme (mit gleicher Schnittstelle) durch die Relationen R_1 und R_2 beschrieben. System R_2 ist eine (Verhaltens-) Verfeinerung *von R_1, notiert durch*

$$R_1 \rightsquigarrow R_2$$

wenn jedes Verhalten von R_2 auch ein Verhalten von R_1 ist:

$$R_2 \subseteq R_1$$

□

Sind die Systeme durch Black-Box Spezifikationen Φ_1 und Φ_2 spezifiziert, so liegt eine Verhaltensverfeinerung vor, wenn

$$\Phi_2 \Rightarrow \Phi_1$$

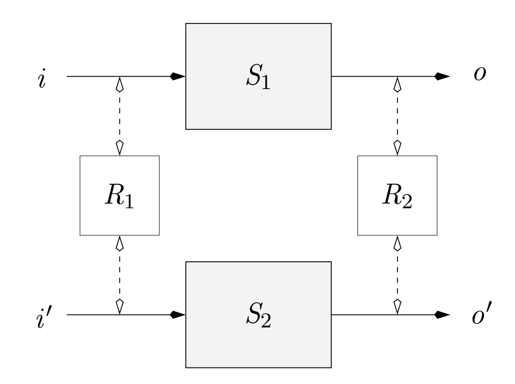


Abbildung 3.9: Schnittstellenverfeinerung

Ein typischer Schritt in der Entwicklung eines Systems ist die Reduktion von Nichtdeterminismus und Unterspezifikation. Dies kann durch eine Verhaltensverfeinerung formalisiert werden. Ist das verfeinerte System aus Unterkomponenten zusammengesetzt, nennt man diesen Schritt auch *strukturelle Verfeinerung.*

3.9.2 Schnittstellenverfeinerung

Die Schnittstellenverfeinerung erlaubt es, die Veränderung der Schnittstelle eines Systems als Verfeinerungsschritt zu formalisieren. So können Anzahl und Typ der Nachrichtenkanäle und die Repräsentation von Nachrichten verändert werden. Prominentes Beispiel ist die Verfeinerung von Nachrichten, die abstrakt durch natürliche Zahlen, implementierungsnah jedoch durch Bitsequenzen dargestellt werden.

Die Schnittstellenverfeinerung wird in Abbildung 3.9 veranschaulicht. Die Nachrichten des abstrakten Systems S_1 werden vermöge der Abstraktions- bzw. Repräsentationsrelationen R_1 und R_2 mit Nachrichten des konkreten Systems S_2 in Beziehung gesetzt. Eine Verfeinerung liegt vor, wenn jedes Verhalten von S_2 nach einer „Übersetzung" mit R_1 und R_2 auch ein Verhalten von S_1 ist.

Definition 3.15 *Schnittstellenverfeinerung*

Seien zwei Systeme S_1 und S_2 mit den Schnittstellen (I, O) bzw. (I', O') gegeben, und zwei Stromrelationen R_1 und R_2 mit den Schnittstellen (I, I') und (O, O'). Das System S_2 ist eine Schnittstellenverfeinerung *von S_1 bezüglich (R_1, R_2), notiert durch*

$$S_1 \overset{(R_1,R_2)}{\rightsquigarrow} S_2$$

wenn gilt

$$\forall\, i', o' \bullet (i', o') \in S_2 \Rightarrow \exists\, i, o \bullet (i, i') \in R_1 \wedge (i, o) \in S_1 \wedge (o, o') \in R_2$$ □

Die Schnittstellenverfeinerung ist eine Verallgemeinerung der Verhaltensverfeinerung: Mit R_1 und R_2 als Identitätsrelation fallen die beiden Verfeinerungsbegriffe zusammen. Die beiden Abstraktions- bzw. Repräsentationsrelationen können wie gewöhnliche Systeme mit den bereits vorgestellten Beschreibungstechniken spezifiziert werden.

In [15, 17, 21] sind Varianten obiger Definitionen angegeben und Kriterien, die die Kompositionalität sicherstellen sowie Fälle von Verfeinerungen ausschließen, die aus praktischer Sicht nicht sinnvoll sind.

Auf eine Darstellung der noch allgemeineren *bedingten Verfeinerung* verzichten wir, da sie in dieser Arbeit keine Rolle spielen wird. Mit ihr ist es möglich, zusätzliche Bedingungen an die Eingaben des Systems zu formulieren, unter denen eine Verfeinerung vorliegt. Eine detaillierte Diskussion findet sich wieder in [21].

3.9.3 Kompositionalität

Eine wichtige Eigenschaft der Verfeinerungsbegriffe im Rahmen einer methodischen Systementwicklung ist ihre *Kompositionalität*: Verfeinert man eine Komponente eines zusammengesetzten Systems, so ergibt sich eine Verfeinerung des Gesamtsystems. Dies ist für die Bewältigung der Komplexität in einem Entwicklungsprozeß wesentlich. Die Verfeinerung der einzelnen Komponenten kann damit unabhängig voneinander geschehen. Die Kompositionalität wird mit folgender Regel ausgedrückt:

$$\frac{S_1 \rightsquigarrow S_1' \qquad S_2 \rightsquigarrow S_2'}{S_1 \otimes S_2 \rightsquigarrow S_1' \otimes S_2'}$$

Im Falle der Schnittstellenverfeinerung ist die Verfeinerung nicht gänzlich unabhängig, da die Veränderung von Kanälen und ihren Nachrichtentypen jeweils eine schreibende und eine lesende Komponente betrifft. Durch die explizite Formalisierung dieser Änderungen mit Hilfe der Abstraktions- und Repräsentationsrelationen R ist der Zusammenhang aber formal dokumentiert, und entsprechende Beweisverpflichtungen ergeben sich aus der folgenden Regel, die in Abbildung 3.10 illustriert ist.

$$\frac{S_1 \overset{(R_I \wedge R_X, R_O \wedge R_Y)}{\rightsquigarrow} S_1' \qquad S_2 \overset{(R_K \wedge R_Y, R_L \wedge R_X)}{\rightsquigarrow} S_2'}{S_1 \otimes S_2 \overset{(R_I \wedge R_K, R_O \wedge R_L)}{\rightsquigarrow} S_1' \otimes S_2'}$$

Die Relationen R müssen gewisse Voraussetzungen erfüllen, um die Kompositionalität zu gewährleisten (wie in [17, 21] begründet). Diese sind hinreichend durch die *Umkehrbarkeit* oder *Linkseindeutigkeit* von R sichergestellt, wenn also $(x, y) \in R \wedge (x', y) \in R \Rightarrow x = x'$ gilt.

3.10 Entwicklungsprozeß

Wie bereits in Abschnitt 3.9 erwähnt, ist es nicht möglich, ein komplexes System in einem Schritt zu entwickeln. Eine formale Methode muß es ermöglichen, ein System auf verschiedenen Abstraktionsebenen darstellen zu können und den Übergang zwischen

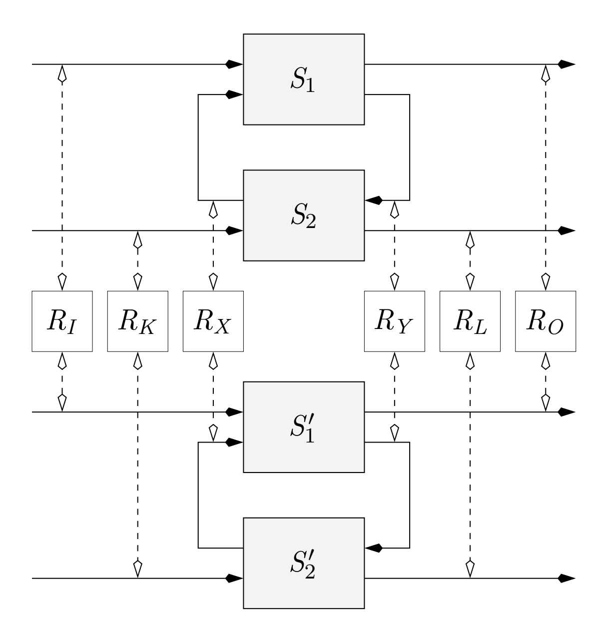


Abbildung 3.10: Schnittstellenverfeinerung eines komponierten Systems

verschiedenen Ebenen formal zu unterstützen. Dabei sollte das Spektrum der Darstellungsmöglichkeiten von abstrakten Systemanforderungen bis hin zu einer konkreten Implementierung reichen.

Als Vorgehensweise kennen wir zwei idealisierte Richtungen: Eine *Top-Down*-Entwicklung führt ausgehend von einer abstrakten Spezifikation schrittweise zu einer lauffähigen Implementierung, während eine *Bottom-Up*-Entwicklung von der Entwicklung von Einzelkomponenten ausgeht, die zielführend zu einem Gesamtsystem zusammengesetzt werden. In der Realität wird man meist eine Mischform wählen, in der man sowohl von abstrakten Anforderungen als auch von bereits implementierten und wiederzuverwendenden Einzelkomponenten ausgehen wird.

Das Ergebnis einer idealisierten Systementwicklung läßt sich vereinfacht als ein Entwicklungsbaum darstellen, wie er in Abbildung 3.11 illustriert ist. Der Unterschied zwischen einem Top-Down- und Bottom-Up-Verfahren liegt dann nur in der Reihenfolge, in welcher der Baum aufgebaut wird. Die Wurzel des Entwicklungsbaumes enthält eine abstrakte Black-Box Spezifikation Φ, die das Verhalten des gesamten Systems beschreibt. Diese Spezifikation wird typischerweise präzisiert durch die Hinzunahme weiterer Anforderungen und das Fällen von Entwurfsentscheidungen zu einer Spezifikation Φ'. Eine strukturelle Verfeinerung hat als Ergebnis eine Reihe von Unterspezifikationen $\Phi_1, \ldots, \Phi_n$. Dazu gehört eine (in der Abbildung nicht dargestellte) Verschaltung, die die Kanäle zwischen den Komponenten und damit die möglichen Interaktionen beschreibt. Die Komponenten selbst können wieder weiter in ihrem Verhalten, ihrer Schnittstelle oder ihrer Struktur verfeinert werden.

Da die Komposition und der Nachweis einer Verfeinerung auf der Ebene der Black-Box Spezifikationen relativ leicht durch einfache Konjunktionen und Implikationen auszu-

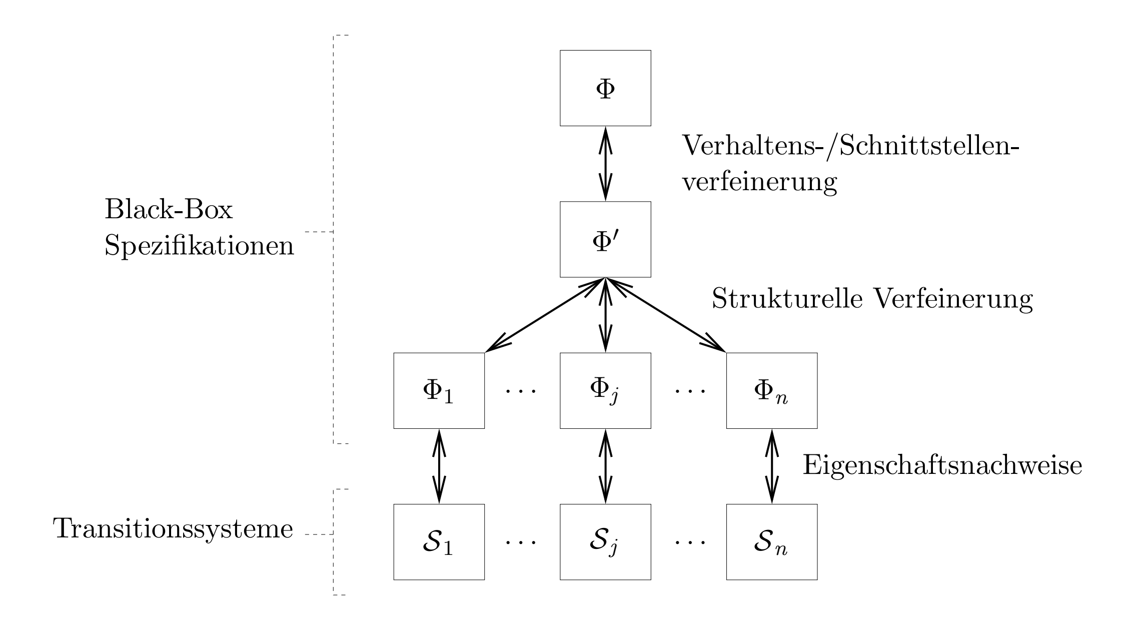


Abbildung 3.11: Idealisierter Entwicklungsprozeß

drücken ist, empfiehlt es sich, die Systembeschreibungen möglichst lange auf diesem Abstraktionsniveau zu halten. Letztendlich sollten die Komponenten aber implementierungsnäher durch Transitionssysteme beschrieben werden. Die Blätter unseres Entwicklungsbaumes enthalten daher Zustandstransitionssysteme, von denen mit den Methoden aus Abschnitt 3.8 gezeigt wird, dass sie tatsächlich die geforderten Eigenschaften aufweisen.

Es sind noch weitere Verfeinerungen auf der Ebene der Transitionssysteme denkbar. Die praktische Anwendbarkeit ist jedoch beschränkt, so dass wir diese hier nicht näher beschreiben, sondern auf [56] verweisen. Auch auf den Übergang von Transitionssystemen zu Implementierungen in konkreten Programmiersprachen, der wie im Werkzeug AutoFocus [6] weitgehend automatisiert erfolgen kann, wird hier nicht weiter eingegangen.

Das Ziel dieser Arbeit ist es, die möglichen Fehler eines Systems zu modellieren und sinnvoll mit ihnen umgehen zu können. Dazu sollen auch in einer bereits durchgeführten Systementwicklung nachträglich noch Fehler berücksichtigt werden. Dies erfordert eine Anpassung des Systementwicklungsbaumes: Wird ein Knoten des Baumes modifiziert, so müssen auch die damit verbundenen Knoten entsprechend modifiziert werden, so dass wieder ein korrekter Systementwicklungsbaum vorliegt. In den Abschnitten 4.5.1 und 4.5.2 sowie in weiten Teilen von Kapitel 5 wird diese Thematik behandelt.

3.11 Zusammenfassung

Neben dem hier vorgestellten Systemmodell von Focus gibt es weitere Systemmodelle, die geeignet sind, verteilte, reaktive Systeme zu modellieren, wie beispielsweise TLA

(*Temporal Logic of Actions*), UNITY oder Prozeßalgebren. Wir haben uns für Focus entschieden, da es folgende Vorteile in sich vereint, die in dieser Kombination alternative Systemmodelle nicht aufweisen:

- Focus bietet ein einheitliches semantisches Grundmodell, auf das die verschiedensten Sichten eines Systems fundiert werden können. Von monolithischen, abstrakten, eigenschaftsorientierten Black-Box Spezifikationen bis hin zu ablauforientierten, implementierungsnahen Beschreibungen von Netzwerken interagierender Komponenten können Systeme auf allen Abstraktionsstufen spezifiziert werden. Alle Beschreibungen können mit Hilfe der Fundierung auf das Strommodell zueinander in Beziehung gesetzt werden. Dies ermöglicht eine durchgängige Systementwicklung ohne Wechsel des semantischen Modells.
- Der Schnittstellenbegriff ermöglicht eine deutliche Trennung zwischen dem System und seiner Umgebung und zwischen den verschiedenen Komponenten eines zusammengesetzten Systems. Die Interaktion zwischen den Systemteilen ist so klar definiert; unerwünschte Seiteneffekte sind bereits durch Anforderungen des Systemmodells ausgeschlossen.
- Eine Konsequenz des Schnittstellenbegriffs liegt in der Kompositionalität des Modells. Damit ist eine modulare Systementwicklung möglich. Die Komponenten eines Systems können demzufolge weitgehend unabhängig voneinander entwickelt werden, da dies automatisch einer Weiterentwicklung des Gesamtsystems gleichkommt. Die Eigenschaften eines Systems sind im wesentlichen unabhängig von seiner Umgebung, und bleiben in jedem Kontext erhalten. Dies unterstützt einen Entwickler in der Handhabung komplexer Systeme.
- Für eine praktische Einsetzbarkeit einer formalen Methodik ist eine Werkzeugunterstützung absolut unerläßlich. Mit AutoFocus und seiner Weiterentwicklung im Rahmen des Projektes Quest [53] steht bereits ein Werkzeug zur Verfügung, mit dem sich reale Systementwicklungen durchführen lassen.

Für die Modellierung von Fehlern stellt dieses Systemmodell daher eine geeignete Grundlage dar um Fehler auf verschiedenen Abstraktionsebenen repräsentieren zu können. Das Modell ermöglicht es, aus der Kenntnis des Komponentenverhaltens auf das Verhalten des Gesamtsystems zu schließen (nach [57] relevant beim analytischen Umgang mit Fehlern). Es ist weiterhin möglich, zwischen internen und externen Fehlern zu unterscheiden, und die Ausbreitung von Fehlern über verschiedene Komponenten hinweg zu untersuchen. Diese Themen werden in den folgenden beiden Kapiteln behandelt.

Im Rahmen von Focus gibt es alternative Modellierungstechniken, die sowohl Zeit als auch Synchronität betreffen. Wir haben uns aus den im folgenden angegebenen Gründen für das *ungezeitete* und *asynchrone* Systemmodell entschieden.

- Im sogenannten *ungezeiteten* Modell wird Zeit nicht modelliert. Es ist damit nicht möglich, das zeitliche Verhalten eines Systems oder einer Komponente *quantitativ* auszudrücken (wobei aber die Kausalität dennoch sichergestellt bleibt). Wir beschränken uns damit auf eine abstrakte Darstellung von Systemen, indem wir alle

Varianten von Verhalten zusammenfassen, die sich nur in ihrem konkreten Ablauf in der Zeit unterscheiden.

Wir wollen uns in dieser Arbeit mit der Modellierung von Fehlern beschäftigen. Auftretende Fehler können im allgemeinen neben einer Veränderung des Verhaltens auch eine Veränderung in der Zeit bewirken. Diese Aspekte sind in einer vollständigen Behandlung von Fehlern selbstverständlich zu berücksichtigen, implizieren aber eine zusätzliche Komplexität in der Behandlung. Für die Erarbeitung der grundlegenden Techniken im Umgang mit Fehlern im Rahmen dieser Arbeit wollen wir uns hier aber auf das ungezeitete Systemmodell beschränken, um die Modellierung von Fehlern zunächst relativ einfach darstellen zu können. Daher fokussieren wir in dieser Arbeit zunächst auf die Auswirkungen von Fehlern ohne Berücksichtigung zeitlicher Aspekte und regen eine weiterführende Untersuchung in Kapitel 6 an.

- Im *asynchronen* Modell erfolgt die Kommunikation zwischen zwei Komponenten so unabhängig wie möglich: Das Senden einer Nachricht ist zu jedem Zeitpunkt möglich und vom Empfangen der Nachricht entkoppelt. Es gibt keine Garantien, wann und ob die Nachricht vom Empfänger gelesen wird. Sichergestellt ist nur, dass der Empfang einer Nachricht nicht *vor* dem Senden einer Nachricht geschieht.

  Da wir verteilte Systeme modellieren wollen, in denen die Nachrichtenübermittlung unterbrochen oder verzögert sein kann, und da wir keine Annahmen über die Übertragungszeiten machen wollen, ist die Wahl des asynchronen Modells angemessen.

FOCUS bietet alle Varianten von Systemmodellen an: Im gezeiteten Modell wird Zeit durch spezielle Nachrichten modelliert, die den Fortschritt der Zeit repräsentieren. Im taktsynchronen Modell wird auf allen Kanälen jeweils „gleichzeitig" genau eine Nachricht gelesen und geschrieben. Ist eine Komponente nicht bereit, eine Nachricht zu lesen, so muß die Nachricht explizit gepuffert werden. In [21] werden die alternativen Modelle, die FOCUS bietet, vorgestellt.

Im nächsten Kapitel werden wir das Systemmodell so erweitern, dass damit die Fehler eines Systems modelliert werden können.

Kapitel 4

Formale Modellierung von Fehlern

Nachdem im vorangegangenen Kapitel das Systemmodell vorgestellt wurde, wollen wir dieses nun um neue Begriffe erweitern, die es uns ermöglichen, mit Fehlern von Systemen explizit umzugehen. Selbstverständlich sind auch Fehler nur ein Teil von Systemen und ihrem Verhalten und könnten aufgrund der Allgemeinheit und Ausdruckskraft des Systemmodells auch ohne die folgenden Begriffe modelliert werden. Allerdings verhelfen vorbereitete Begriffsbildungen zu einem besseren Verständnis und können einen bewußteren Umgang mit Fehlern und ihren spezifischen Eigenschaften ermöglichen.

Fehler können nur *relativ* zu einer als korrekt und damit fehlerfrei akzeptierten Fassung eines Systems definiert werden. Liegt ein korrektes System S und seine fehlerbehaftete Version S' vor, ist der Fehler als der *Unterschied* zwischen diesen beiden Fassungen beschrieben, wenn auch nur sehr implizit. Der Fehler selbst wäre so noch nicht direkt formal greifbar. Der Unterschied kann alle Merkmale eines Systems betreffen, wobei Unterschiede im Verhalten im Vordergrund stehen. Wir werden noch genauer differenzieren zwischen Unterschieden statischer Systemanteile, Unterschiede in der Schnittstelle und Unterschiede im Verhalten eines Systems.

Wir interpretieren Fehler eines Systems explizit als die *Modifikation*, die aus der fehlerfreien Version die fehlerbehaftete macht. Eine derartige Modifikation erlaubt es damit, von einer idealisierten Sicht auf ein System zu einer realistischen und damit implementierungsnäheren Sicht übergehen zu können.

Die formale Definition von Fehlern eines Systems erlaubt eine Unterscheidung von Systemverhalten in drei disjunkte Klassen, die in Abbildung 4.1 dargestellt sind. In der Mehrzahl typischer formaler Ansätze wird nur das intendierte Verhalten eines Systems untersucht und spezifiziert. Bei der Berücksichtigung von Fehlern wird zusätzliches Systemverhalten in die formale Behandlung mit einbezogen, das vom Normalverhalten abweicht. Allerdings wird dabei nicht *jedes* abweichende Verhalten berücksichtigt, sondern eine Unterscheidung gemacht zwischen Abweichungen, mit deren Auftreten man umgehen möchte und den restlichen Abweichungen, die man entweder für unmöglich oder für unwahrscheinlich genug hält, um sie in Betracht ziehen zu müssen. Auch andere for-



Abbildung 4.1: Verhaltensklassen

male Ansätze zur Fehlermodellierung klassifizieren Verhalten in diese drei Typen [59]. Mit dem Formalismus, den wir in dieser Arbeit präsentieren, ist es möglich, eine scharfe Trennung zwischen diesen drei Klassen zu ziehen. Wir können damit Abweichungen vom Normalverhalten, die formal zu untersuchen und für die etwaige Gegenmaßnahmen zu ergänzen sind, exakt spezifizieren.

In den folgenden beiden Abschnitten beschreiben wir, was Modifikationen von Systemen sind und wie sie formal repräsentiert werden – unterschieden nach Modifikationen der Schnittstelle eines Systems (Abschnitt 4.1) und Modifikationen des Verhaltens (Abschnitt 4.2). Wir nutzen in Abschnitt 4.3 und 4.4 die formalen Fundierungen zur Definition von sonst meist nur informell beschreibbaren Begriffen aus dem Umfeld der Fehler und Fehlertoleranz. In Abschnitt 4.5 untersuchen wir formale Eigenschaften von Modifikationen wie ihre Kombination und ihre Auswirkung über verschiedene Abstraktionsstufen hinweg. Fehler können nicht nur innerhalb eines Systems auftreten, sondern ebenso in der Umgebung eines Systems. Dieses Thema behandeln wir in Abschnitt 4.6. Wir schließen in 4.7 mit einer Diskussion.

Im darauffolgenden Kapitel 5 werden wir die Verwendungsmöglichkeiten des dargestellten erweiterten Formalismus eingehender diskutieren.

4.1 Modifikation der Schnittstelle

Wie in Kapitel 3 beschrieben, definiert man ein System durch zwei wesentliche Teile: seine Schnittstelle und sein Verhalten. Wir wollen beide Teile unabhängig voneinander modifizieren können, also die Schnittstelle verändern, und dabei das Verhalten *weitgehend* unverändert lassen, und dann das Verhalten modifizieren, ohne dabei gleichzeitige Änderungen in der Systemschnittstelle berücksichtigen zu müssen. Diese Trennung erleichtert einerseits die formale Fehlermodellierung und entspricht auch dem Wunsch, bei einer Systementwicklung unterschiedliche Aspekte weitgehend getrennt zu halten und damit die Komplexität einzelner Entwicklungsschritte im Rahmen zu halten.

Im Laufe einer Systementwicklung ist es möglich, dass nicht von vornherein absehbar ist, welche Kanäle mit welchen Typen zwischen dem System und seiner Umgebung bzw. zwischen den Komponenten eines Systems notwendig sind. So kann in einer abstrakten, idealisierten Spezifikation, in der Fehlerfälle noch gänzlich unberücksichtigt sind, keine Möglichkeit vorgesehen sein, um Fehlermeldungen zu übertragen oder um Reaktionen auf Fehlermeldungen (beispielsweise einen Neustartbefehl) auszugeben. Eine Modifikati-

on der Schnittstelle muss also im Rahmen eines Entwicklungsprozesses möglich gemacht werden. In diesem Abschnitt stellen wir dazu die formalen Grundlagen bereit.

Wir betrachten nur Modifikationen der Schnittstelle zur Entwicklungszeit des Systems, also im Rahmen einer Systementwicklung. Veränderungen der Schnittstelle zur Laufzeit des Systems, wie dies in mobilen und dynamischen Systemen [32, 30, 29] geschieht, sind nicht Gegenstand unserer Untersuchungen.

Wir gehen im folgenden – wie in Abschnitt 3.3 – von einem System S mit der Schnittstelle (I, O) aus, wobei

$$I = \{i_1, \ldots, i_n\}$$
$$O = \{o_1, \ldots, o_m\}$$

Das Verhalten von S sei durch eine Relation R gegeben. Wir unterscheiden in den nächsten beiden Abschnitten die *Erweiterung* der Schnittstelle von der *Einschränkung* der Schnittstelle. Während wir eine Erweiterung als Entwicklungsschritt formal unterstützen, werden wir die Einschränkung (unter Angabe von Gründen) nicht zulassen.

4.1.1 Erweiterung der Schnittstelle

Eine Erweiterung der Schnittstelle eines Systems kann sich in vier Varianten ausprägen. Diese im folgenden kurz beschriebenen vier Varianten lassen sich durch Beispiele motivieren, die ihre Relevanz in konkreten Entwicklungsszenarien zeigen.

- Die Schnittstelle wird um einen *neuen Eingabekanal* beliebigen Typs ergänzt. Will man ein System um neue Leistungsmerkmale erweitern, so muss dazu im allgemeinen die Schnittstelle erweitert werden, damit die neuen Merkmale auch nutzbar werden. Beispiele sind Signale zum Rücksetzen oder Neustarten eines Systems oder Fehlermeldungen anderer Komponenten, auf die ein System zu reagieren hat.

- Der *Typ* eines bestehenden Eingabekanals wird erweitert, so dass über ihn neue Nachrichten empfangen werden können. Diese Form der Schnittstellenerweiterung motiviert sich erneut durch den Wunsch nach einer erweiterbaren Funktionalität mit zu ergänzenden neuen Nachrichten auf einem existierenden Kanal. Auch eine Verallgemeinerung einer Systemfunktionalität kann eine Erweiterung des Kanaltyps bedingen: Soll beispielsweise eine Komponente nicht nur natürliche, sondern ganze Zahlen verarbeiten, entspricht dies neuen Nachrichten auf einem bereits existierenden Kanal.

- Die Schnittstelle wird um einen *neuen Ausgabekanal* ergänzt. Sollen beispielsweise zusätzliche, im System vorhandene Informationen oder errechnete Ergebnisse ausgegeben oder auch Fehlermeldungen nach außen sichtbar gemacht werden, ist die Einführung neuer Ausgabekanäle nötig.

- Der *Typ* eines Ausgabekanals wird erweitert. Wiederum können neue Nachrichten, die ausgeben werden sollen, nicht nur auf einem dedizierten Kanal, sondern auch auf bereits existenten Kanälen verschickt werden.

Wir nennen eine Schnittstelle $(\hat{I}, \hat{O})$ eine Erweiterung einer Schnittstelle (I, O), wenn neue Kanäle ergänzt werden oder der Typ existierender Kanäle erweitert wird.

Definition 4.1 *Erweiterung der Schnittstelle*

Die Schnittstelle $(\hat{I}, \hat{O})$ *heißt* Erweiterung der Schnittstelle (I, O), *wenn gilt*

$$\hat{I} \supseteq \{\hat{i} \mid i \in I\}$$
$$\hat{O} \supseteq \{\hat{o} \mid o \in O\}$$
$$\forall\, i \in I \bullet \mathit{type}(i) \subseteq \mathit{type}(\hat{i})$$
$$\forall\, o \in O \bullet \mathit{type}(o) \subseteq \mathit{type}(\hat{o})$$

□

Die erweiterten Kanäle werden zur Unterscheidung von den ursprünglichen Kanälen mit $\hat{}$ markiert, um die Typerweiterung zwischen den alten und den neuen Kanälen ausdrücken zu können. Die Kanalmenge $\hat{I}$ (bzw. $\hat{O}$) enthält also alle Kanalnamen von I (bzw. O), die darin aber markiert sind und möglicherweise noch weitere, neue Kanalnamen.

Beispiel 4.1 Schnittstellenerweiterung von *Merge*

Die Schnittstelle der Komponente *Merge* ist

$$I = \{i_1, i_2\}$$
$$O = \{o\}$$

mit

$$\mathit{type}(i_1) = \mathit{DATA}_1$$
$$\mathit{type}(i_2) = \mathit{DATA}_2$$
$$\mathit{type}(o) = \mathit{DATA}_1 \cup \mathit{DATA}_2$$

Nehmen wir an, wir wollen den Typ der Eingabekanäle um ein weiteres Datum $\mathit{high} \notin \mathit{DATA}_1 \cup \mathit{DATA}_2$ ergänzen, das anzeigt, dass das folgende Datum (durch eine später zu ergänzende entsprechende Modifikation des Verhaltens) mit höherer Priorität eingemischt werden soll. Ferner wollen wir die Komponente um einen neuen Ausgabekanal erweitern, auf dem Fehlermeldungen gesendet werden können. Die neue Schnittstelle

$$\hat{I} = \{\hat{i_1}, \hat{i_2}\}$$
$$\hat{O} = \{\hat{o}, \hat{f}\}$$

mit

$$\mathit{type}(\hat{i_1}) = \mathit{DATA}_1 \cup \{\mathit{high}\}$$
$$\mathit{type}(\hat{i_2}) = \mathit{DATA}_2 \cup \{\mathit{high}\}$$
$$\mathit{type}(\hat{o}) = \mathit{DATA}$$

ist eine Erweiterung gemäß obiger Definition. Der Typ des Kanals f bleibt dabei noch unspezifiziert. □

Eine Erweiterung der syntaktischen Schnittstelle soll unabhängig von einer Verhaltensänderung geschehen. Da sich die Typen der Kanäle verändert haben, muss auch die verhaltensbeschreibende Relation angepaßt werden, aber in einer Weise, die das *Verhalten* nicht wesentlich verändert. Wir müssen nun also definieren, welche an die neue Schnittstelle angepaßte Relation wir als das „im wesentlichen gleiche“ Verhalten interpretieren. Ausgehend von der existierenden, auf die unmodifizierte Schnittstelle bezogene Relation soll die angepaßte Verhaltensrelation folgende Eigenschaften aufweisen:

- Die Eingabe über neue Kanäle soll ignoriert werden, also keinen Einfluß auf das an den alten Ausgabekanälen sichtbare Ausgabeverhalten haben. Das Verhalten darf nur abhängen von den Eingaben auf den ursprünglichen, bereits vorhandenen Kanälen.
- Neue Nachrichten auf bereits vorhandenen Kanälen werden ignoriert.
- Auf neuen Ausgabekanälen dürfen beliebige Nachrichten (des passenden Typs) gesendet werden. Diese dürfen zwar auch von den Eingaben an neuen Kanälen abhängen, müssen es aber nicht.
- Auf typerweiterten Ausgabekanälen werden nur Nachrichten vom ursprünglichen Typ gesendet.

Ein derartiges in seiner Schnittstelle erweitertes und im Verhalten angepaßtes System kann damit in der gleichen Umgebung eingesetzt werden wie das ursprüngliche System, ohne ein neues Verhalten zu erzeugen. Die genannten Anforderungen sind in der folgenden Definition reflektiert.

Definition 4.2 *Verhaltensgleichheit bei veränderter Schnittstelle*
Sei durch die Relation R das Verhalten eines Systems mit der Schnittstelle (I, O) spezifiziert, d.h.

$$\begin{aligned} R \;\subseteq\;& \mathit{type}(i_1)^\omega \times \ldots \times \mathit{type}(i_n)^\omega \\ & \times \\ & \mathit{type}(o_1)^\omega \times \ldots \mathit{type}(o_m)^\omega \end{aligned}$$

Sei ferner $(\hat{I}, \hat{O})$ eine erweiterte Schnittstelle mit k neuen Eingabe- und l neuen Ausgabekanälen. Die Relation $\hat{R}$ vom Typ

$$\begin{aligned} \hat{R} \;\subseteq\;& \mathit{type}(\hat{i_1})^\omega \times \ldots \times \mathit{type}(\hat{i_n})^\omega \times \mathit{type}(\hat{i}_{n+1})^\omega \times \ldots \times \mathit{type}(\hat{i}_{n+k})^\omega \\ & \times \\ & \mathit{type}(\hat{o}_1)^\omega \times \ldots \times \mathit{type}(\hat{o}_m)^\omega \times \mathit{type}(\hat{o}_{m+1})^\omega \times \ldots \times \mathit{type}(\hat{o}_{m+l})^\omega \end{aligned}$$

heißt die verhaltensgleiche *Relation zu R und ist definiert durch*

$$\begin{aligned} &(x_1, \ldots, x_{n+k}, y_1, \ldots, y_{m+l}) \in \hat{R} \quad \Leftrightarrow_{def} \\ &\quad (\mathit{type}(i_1) \circledS x_1, \ldots, \mathit{type}(i_n) \circledS x_n, y_1, \ldots, y_m) \in R \end{aligned}$$

□

In der Definition wurden entgegen der Konvention die Kanalnamen und die Nachrichtenströme nicht mit den gleichen Variablen bezeichnet, da dies zu verwirrenden Ausdrücken wie $type(i)Ⓢi$ geführt hätte. Stattdessen wurden die neuen freien Variablen x und y zur Repräsentation der Ein- und Ausgabeströme verwendet. Wir veranschaulichen die Definition an einem Beispiel.

Beispiel 4.2 Verhaltensgleichheit von *Merge*

Betrachten wir erneut die Komponente *Merge* mit der veränderten Schnittstelle. Das konkrete Verhalten

$$\hat{i}_1 = \langle d_1, high, d_1', d_1'' \rangle$$
$$\hat{i}_2 = \langle d_2, d_2', high \rangle$$
$$\hat{o} = \langle d_1, d_2, d_1', d_1'', d_2' \rangle$$
$$\hat{f} = \langle \ldots \rangle$$

ist ein erlaubtes Verhalten, da durch

$$i_1 = \langle d_1, d_1', d_1'' \rangle$$
$$i_2 = \langle d_2, d_2' \rangle$$
$$o = \langle d_1, d_2, d_1', d_1'', d_2' \rangle$$

ein Verhalten der *Merge*-Komponente (mit unveränderter Schnittstelle) beschrieben ist. Auf dem neuen Ausgabekanal f dürfen beliebige Nachrichten gesendet werden. Das Ignorieren neuer Nachrichtentypen in den Eingaben sowie beliebiger Ausgaben über neue Ausgabeströme ist in diesem Beispiel also unmittelbar erkennbar. Hätten wir auch einen neuen Eingabekanal ergänzt, so hätten auch auf diesem beliebige Nachrichtenströme empfangen werden können. □

Es bleibt zu klären, ob die so definierte Erweiterung der Schnittstelle die in Abschnitt 3.4 geforderten Eigenschaften der Verhaltensrelation nach Realisierbarkeit erhält. Die Linkstotalität bleibt offensichtlich erhalten, wie leicht gezeigt werden kann:

Proposition 4.1 *Erhalt der Linkstotalität*

Seien $x_1, \ldots, x_{n+k}$ Eingabeströme, und sei R eine Verhaltensspezifikation, also linkstotal. Die Ströme $type(i_1)Ⓢx_1, \ldots, type(i_n)Ⓢx_n$ stellen typkorrekte Eingaben an ein durch R beschriebenes System dar, so dass aufgrund der Linkstotalität von R passende Ströme $y_1, \ldots, y_n$ existieren, die eine mögliche Ausgabe des Systems repräsentieren. Ergänzt man diese um beliebig wählbare $y_{n+1}, \ldots, y_{n+l}$, so hat man (zu einer beliebigen Eingabe) eine Ausgabe gefunden, die bezüglich $\hat{R}$ paßt. □

Die Kausalität bleibt auch erhalten, wenn man die Aussage auf die Ausgabekanäle beschränkt, die nicht neu ergänzt wurden, wie die folgende Proposition zeigt. Auf den neuen Kanälen sind beliebige Ausgaben möglich, also auch nicht-monotones Verhalten. Da ein neuer Kanal üblicherweise zweckgebunden eingeführt wird, wird das Verhalten in weiteren Entwicklungsschritten (durch eine Verhaltensmodifikation) präziser spezifiziert und damit auch kausal. Wir haben daher in Definition 4.2 auf eine explizite Sicherstellung der Kausalität verzichtet.

Proposition 4.2 *Partieller Erhalt der Kausalität*

Gelte $x_j \sqsubseteq x'_j$ *für alle* $j \in \{1, \ldots, (n+k)\}$, *und*

$$(x_1, \ldots, x_n, x_{n+1}, \ldots x_{n+k};\ y_1, \ldots, y_m, y_{m+1}, \ldots x_{m+l}) \in \hat{R}$$
$$(x'_1, \ldots, x'_n, x'_{n+1}, \ldots x'_{n+k};\ y'_1, \ldots, y'_m, y'_{m+1}, \ldots x'_{m+l}) \in \hat{R}$$

Aufgrund der Definition verhaltensgleicher Relationen gilt also

$$(type(i_1)\circledS x_1, \ldots, type(i_n)\circledS x_n, y_1, \ldots, y_m) \in R$$
$$(type(i_1)\circledS x'_1, \ldots, type(i_n)\circledS x'_n, y'_1, \ldots, y'_m) \in R$$

Da gilt, dass $x \sqsubseteq x' \Rightarrow M\circledS x \sqsubseteq M\circledS x'$, *folgt daraus mit der Kausalität von* R *direkt*

$$\forall\, j \in \{1, \ldots, m\} \bullet y_j \sqsubseteq y'_j$$

□

Die Beziehung zwischen den durch R und R' beschriebenen Systemen stellt eine *Schnittstellenverfeinerung* im Sinne von Abschnitt 3.9.2 dar. Die Umsetzungsrelationen R_I bzw. R_O ergeben sich unmittelbar aus Definition 4.2. Sie sind entsprechend Abschnitt 3.9.3 umkehrbar, so dass die in diesem Abschnitt präsentierte Erweiterung der Schnittstelle *kompositional* erfolgen kann.

4.1.2 Einschränkung der Schnittstelle

Die Schnittstelle eines Systems können wir einschränken, indem wir Kanäle entfernen oder den Typ von Kanälen reduzieren. Dies kann aber zu Problemen führen, wenn dies – wie im Fall der Schnittstellenerweiterung – auf eine Art und Weise geschehen soll, die das Verhalten unverändert läßt.

Ein problematischer Fall ist die Beschränkung von Ausgabekanälen. Hat ein unmodifiziertes System auf eine bestimmte Eingabe mit einer oder mehreren Ausgabenachrichten reagiert, die nun durch die Modifikation der Schnittstelle nicht mehr möglich sein können, ist die Linkstotalität der Verhaltensrelation gefährdet. Ist das System deterministisch, so läßt sich in diesem Fall keine allgemein definierbare, sinnvolle Reaktion des Systems mehr finden.

Doch auch das Entfernen von Eingabekanälen kann zu Komplikationen führen, für die sich keine sinnvolle, allgemeine, verhaltensneutrale Anpassung der Verhaltensrelation finden läßt. War der Fortschritt einer Systemaktivität beispielsweise von einem stetigen Empfang von Nachrichten über einen bestimmten Kanal abhängig, und wird dieser Kanal entfernt, so bleibt die Frage offen, wie das Verhalten nun angepaßt werden soll: Soll die Aktivität von diesem Kanal unabhängig gemacht werden oder soll das System keinerlei Ausgaben mehr machen?

Die Typeinschränkung auf den Eingabekanälen stellt dagegen ein unwesentlicheres Problem dar, da die verhaltensbeschreibende Relation vollkommen unverändert bleiben kann. Die Einschränkung bedeutet informell, dass gewisse Eingaben nicht mehr auftreten, auch wenn für sie eine Reaktion definiert wäre. Für die noch immer möglichen, verbleibenden Eingaben bleibt die Ausgabe natürlicherweise definiert und unverändert.

Diese Probleme legen die in dieser Arbeit befolgte Konsequenz nahe, auf eine *Einschränkung der Schnittstelle zu verzichten.* Im Gegensatz zur Erweiterung der Schnittstelle, die durchaus für eine Anpassung eines Systems an veränderte Neben- und Umgebungsbedingungen nötig ist, kann auch ohne eine formale Unterstützung der Einschränkung von Systemschnittstellen eine Komponente mit allen Freiheiten beliebig weiterentwickelt werden: Bestimmte Kanäle existieren in diesem Fall weiterhin, treten also in Form von Platzhaltern in den Tupeln auf, auch wenn sie nicht mehr in die Funktionalität eingehen, also ihre Variablen nicht mehr in den Formeln für Verhaltensbeschreibungen oder den Eingabemustern von Transitionssystemen erscheinen.

Die Beschränkung der Eingabe an ein System entspricht einer stärkeren Bedingung, die die Systemumgebung erfüllt. Unter stärkeren Umgebungsbedingungen weist ein System im allgemeinen auch stärkere Eigenschaften auf, für die es im Rahmen der Verifikation möglich sein muss, dass diese formal gezeigt werden können. Dies ist aber auch ohne eine formale Schnittstellenbeschränkung möglich: Durch die Formulierung einer Umgebungsannahme entsprechend Abschnitt 3.5 läßt sich zusätzliches Wissen formulieren und in der Verifikation nutzen.

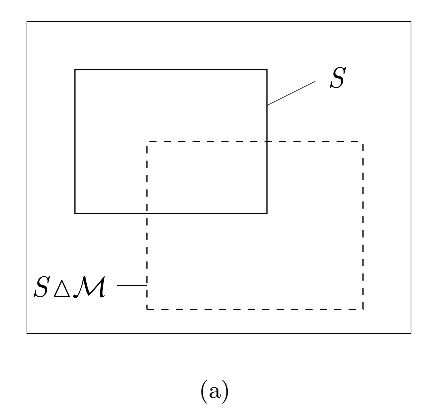
4.2 Modifikation des Verhaltens

Wir behandeln nun in diesem Abschnitt die *Modifikation* des System*verhaltens*. Wir bezeichnen eine Verhaltensmodifikation mit $\mathcal{M}$, die im folgenden in den verschiedenen Beschreibungstechniken jeweils formal definiert wird. Eine Modifikation $\mathcal{M}$ kann auf ein System S angewendet werden und beschreibt den *Unterschied* zwischen dem ursprünglichen System und seiner veränderten Variante. Das Resultat ist das modifizierte System, und wird notiert als

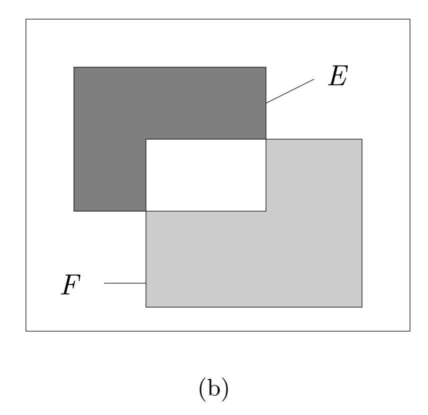
$$S \triangle \mathcal{M} \quad \text{(zu lesen als „}S\text{ modifiziert durch }\mathcal{M}\text{“)}$$

Eine Modifikation $\mathcal{M}$ verändert ein System S auf zwei Weisen: Es *entfernt* einerseits gewisse Anteile des Systems, und fügt andererseits andere – im allgemeinen neue – Teile dem System hinzu. Diese Veränderungen können die verschiedensten Aspekte eines Systems betreffen wie Eigenschaften, interne Transitionen oder das Black-Box-Verhalten.

Wir illustrieren mit Abbildung 4.2(a) die Auswirkung einer Modifikation, die das Verhalten eines Systems S modifiziert. Die Gesamtfläche stelle die Menge aller möglichen Verhalten dar. Darin wird das Verhalten eines Systems S durch eine Teilmenge repräsentiert. Das Verhalten des modifizierten Systems $S \triangle \mathcal{M}$ stellt nun eine andere Teilfläche dar, die nur noch einen Teil des Verhaltens von S umfaßt, dafür aber neue Flächenteile dazugewinnt. Der Unterschied kann durch die Angabe von zwei in Abbildung 4.2(b) illustrierten Angaben explizit dargestellt werden: Der dunkel schattierte Teil E ist der Teil des Verhaltens, den S durch die Modifikation verliert. Der durch F repräsentierte grau schattierte Teil stellt die neuen Anteile des Verhaltens dar. Diese Idee, die Veränderung durch Angabe von neuen und zu entfernenden Teilen explizit zu machen, wird sich in den folgenden Formalisierungen von Modifikationen widerspiegeln.



(a)



(b)

Abbildung 4.2: Verhaltensmodifikation

Im folgenden geben wir an, wie konkrete Verhaltensmodifikationen in den in Kapitel 3 vorgestellten Beschreibungstechniken notiert werden können. Wir behandeln die relationale Sicht, die Black-Box Ebene, die Sicht der Transitionssysteme und sogenannte Modifikationskomponenten. Die Zusammenhänge von Modifikationen, die mittels verschiedener Techniken und damit auf verschiedenen Abstraktionsebenen angegeben werden, sind in Abschnitt 4.5.3 beschrieben.

Wir greifen für die Beispiele dieses Abschnitts wieder die Komponente *Merge* aus Kapitel 3 auf, die zwei Eingabeströme zusammenmischt. Wir werden mit den im folgenden dargestellten Beschreibungstechniken immer die gleiche Modifikation beschreiben, die eine Abweichung vom eigentlich intendierten Systemverhalten darstellt: Das System soll einen der beiden Kanäle ignorieren, und daher nur die empfangenen Nachrichten des anderen Kanals auf dem Ausgabekanal wieder ausgeben. Die Auswahl des ignorierten Kanals soll dabei nichtdeterministisch erfolgen.

4.2.1 Modifikationen von Relationen

Das Verhalten wird semantisch als eine Relation zwischen Ein- und Ausgabeströmen repräsentiert, also durch die Menge aller Stromtupel, die in dieser Relation stehen.

Diese Menge kann nur auf zweierlei Arten modifiziert werden: Einerseits durch die *Hinzunahme* weiterer Stromtupel, die zusätzliche Freiheiten im Verhalten des Systems bedeuten, andererseits durch das *Entfernen* von Stromtupeln. Da die Linkstotalität nicht gefährdet werden darf, muss natürlich für jede mögliche Eingabe noch immer ein Stromtupel in der Relation erhalten bleiben.

Diejenigen Stromtupel, die ein modifiziertes System zusätzlich enthält, fassen wir in einer Menge F zusammen. Die Menge der Stromtupel, die das modifizierte System nicht mehr enthält, nennen wir E.

Die Wahl des Bezeichners F orientiert sich am Begriff des *Fehlers* oder auch an *fault* und *failure*: (Transiente) Fehler in einem System sind meist unerwünschte Ausprägungen des

Verhaltens, die ein System zusätzlich aufweisen kann, also ein erweiterter Nichtdeterminismus, der genau durch die Menge F beschrieben wird. Das Entfernen von Teilen des Verhaltens geht mit einer höheren Priorisierung des verbleibenden Verhaltens einher, was einem *Exception*-Mechanismus ähnelt. Dies wird illustriert durch die Wahl des Bezeichners E.

Definition 4.3 *Modifikation von Relationen*

Sei R eine Relation über der Schnittstelle (I, O), und seien auch E und F Relationen vom gleichen Typ, also

$$R, E, F \subseteq type(i_1)^\omega \times \ldots \times type(i_n)^\omega \times type(o_1)^\omega \times \ldots type(o_m)^\omega$$

Die durch

$$\mathcal{M} = (E, F)$$

definierte Modifikation modifiziert das System gemäß

$$R \triangle \mathcal{M} \stackrel{df}{=} (R \setminus E) \cup F$$

□

Die Ausdruckskraft dieser Modifikationen ist hinreichend allgemein, wie leicht einzusehen ist: Eine Relation R kann zu einer beliebigen Relation R' modifiziert werden mittels der Modifikation $\mathcal{M} = (R, R')$. Eine neutrale Modifikation, die keine Änderungen bewirkt (für die die Bezeichnung Modifikation also nicht mehr wirklich zutrifft) wird durch $(\varnothing, \varnothing)$ dargestellt. Die Modifikation mit F als der All-Relation wandelt jedes System R um in ein System mit beliebigem, also chaotischem Verhalten.

Wir fordern in der Definition weder, dass die Stromtupel aus E wirklich in R enthalten sind, noch dass die Ströme in F nicht bereits in R liegen. Es bleibt also formal gestattet, nicht vorhandene Stromtupel zu „entfernen“ und bereits enthaltene Stromtupel erneut in die Menge R aufzunehmen. Damit werden unnötige Einschränkungen bei der Formulierung von Modifikationen in konkreten Beschreibungstechniken vermieden. Wir lassen hier auch zu, dass Ströme durch E entfernt und durch F wieder ergänzt werden. In Abschnitt 4.5.1 werden wir allerdings die Disjunktheit der Mengen E und F zwischen verschiedenen Modifikationen fordern, um die Kombination von Modifikationen zu erleichtern.

Beispiel 4.3 Da das modifizierte System $Merge \triangle \mathcal{M}$ nur noch die Nachrichten eines der beiden Eingabekanäle ausgeben soll, müssen wir alle Ströme entfernen, in denen auf o Nachrichten von Kanal i_1 *und* i_2 vorkommen. Das modifizierte System zeigt kein neues Verhalten, so dass keine neuen Stromtupel ergänzt werden müssen. Wir definieren die Modifikation also zu

$$\mathcal{M} = (E, \varnothing)$$

wobei E alle Tupel enthält, deren Ausgabeströme sowohl Daten mit dem Index 1 als auch mit dem Index 2 enthalten:

$$\begin{array}{rl} E = \{ & (\langle d_1, d_1', d_1'', \ldots\rangle, \langle d_2, d_2', d_2'', \ldots\rangle;\ \langle d_1, d_2, \ldots\rangle) \\ & (\langle d_1, d_1', d_1'', \ldots\rangle, \langle d_2, d_2', d_2'', \ldots\rangle;\ \langle d_2, d_1, \ldots\rangle) \\ & (\langle d_1, d_1', d_1'', \ldots\rangle, \langle d_2, d_2', d_2'', \ldots\rangle;\ \langle d_1, d_1', d_2, \ldots\rangle) \\ & (\langle d_1, d_1', d_1'', \ldots\rangle, \langle d_2, d_2', d_2'', \ldots\rangle;\ \langle d_2, d_2', d_1, \ldots\rangle) \\ & \ldots \\ \} & \end{array}$$

□

Das Beispiel macht (erneut) deutlich, dass die explizite Angabe von Relationen durch Auflistung nicht geeignet ist, um konkrete Systeme und ihre Modifikationen zu definieren. Wir geben im folgenden geeignetere Beschreibungtechniken an.

4.2.2 Modifikationen von Black-Box Spezifikationen

Eine Black-Box Spezifikation Φ kann im wesentlichen auf zwei Arten modifiziert werden: Die durch Φ beschriebene Eigenschaft kann durch Hinzunahme weiterer Eigenschaften *verstärkt* werden, und sie kann durch Anbieten von alternativen Eigenschaften *abgeschwächt* werden.

Die Verstärkung kann durch eine Konjunktion mit den neuen Eigenschaften beschrieben werden, die wir als Φ_F bezeichnen. Die neuen Eigenschaften sind selbst wieder eine Black-Box Spezifikation bei unveränderter Systemschnittstelle. Die Abschwächung wird entsprechend durch eine Disjunktion mit einer Formel Φ_E angegeben. Wir definieren:

Definition 4.4 *Modifikation von Black-Box Spezifikationen*

Seien Φ, Φ_E und Φ_F Black-Box Spezifikationen zur Schnittstelle (I, O). Die Wirkung der durch

$$\mathcal{M} = (\Phi_E, \Phi_F)$$

definierten Modifikation auf die Black-Box Spezifikation Φ ist definiert durch

$$\Phi \triangle (\Phi_E, \Phi_F) \overset{df}{=} (\Phi \wedge \Phi_E) \vee \Phi_F$$

□

Die neutrale Modifikation wird offensichtlich durch das Paar $(\mathsf{true}, \mathsf{false})$ beschrieben, da $\Phi = (\Phi \wedge \mathsf{true}) \vee \mathsf{false}$. Die Modifikationen von Φ zu einem beliebigem Ψ kann ausgedrückt werden durch die Modifikation $\mathcal{M} = (\neg\, \Phi, \Psi)$, oder durch ein $\mathcal{M} = (\Phi_E, \Psi)$ mit $\Phi_E \Rightarrow \neg\, \Phi)$, begründet durch die Tautologie $\Psi = (\Phi \wedge \neg\, \Phi) \vee \Psi$. Die Modifikation zu chaotischem Verhalten läßt sich mit der Modifikation (Φ_E, true) mit beliebigem Φ_E beschreiben.

Die Formel Φ_E definiert eine Menge von Stromtupeln, die diese Formel erfüllen. Diese werden aber nicht aus dem Verhalten des modifizierten Systems entfernt, wie dies bei der Modifikation von Relationen geschieht. Eine unmittelbarere Analogie wäre gegeben, hätte man die Modifikation mit einer Negation durch $(\Phi \wedge \neg\, \Phi_E) \vee \Phi_F$ definiert. Wir haben Definition 4.4 aber so gewählt, dass sich in Φ_E gut die *zusätzlichen* Eigenschaften des veränderten Systems erfassen lassen. Dies unterstützt eher die praktische Verwendung von Modifikationen, wenn die Systemeigenschaften verstärkt werden sollen.

Die Modifikation von Black-Box Spezifikationen ist verträglich mit der Verfeinerung, wie leicht einzusehen ist:

Proposition 4.3 *Es gilt*

$$\frac{\Phi_1 \rightsquigarrow \Phi_2}{\Phi_1 \triangle \mathcal{M} \rightsquigarrow \Phi_2 \triangle \mathcal{M}}$$

Aus $(\Phi_2 \wedge \Phi_E) \vee \Phi_F$ *folgt unter der Voraussetzung* $\Phi_2 \Rightarrow \Phi_1$ *sofort* $(\Phi_1 \wedge \Phi_E) \vee \Phi_F$.

□

Wir beschreiben die Modifikation aus Beispiel 4.3 erneut im Rahmen von Black-Box Spezifikationen. Dabei wird deutlich, dass zwei semantisch äquivalente Modifikationen in verschiedenen Beschreibungstechniken sehr unterschiedlich formuliert werden können.

Beispiel 4.4 Die Komponente *Merge* soll nur noch Nachrichten von i_1 oder i_2 auf o ausgeben. Die Disjunktion dieser beiden Fälle scheint eine zusätzliche Eigenschaft als Verstärkung der Black-Box Sicht mittels Φ_E darzustellen. Konjugiert man aber $o = i_1 \vee o = i_2$ zur Black-Box Spezifikation $D_1 \circledS o = i_1 \wedge D_2 \circledS o = i_2$, so impliziert dies, dass einer der Eingabeströme i_1 oder i_2 leer sein müßte, was der Linkstotalität widerspricht. Um diesen Konflikt zu vermeiden, wählen wir die radikale Lösung durch Verwerfen der ursprünglichen Spezifikation mit $\Phi_E = \mathsf{false}$, und spezifizieren das neue Verhalten vermöge Φ_F. Unsere gesuchte Modifikation läßt sich also definieren als

$$\mathcal{M} \quad \equiv \quad (\mathsf{false}\ ,\ o = i_1 \vee o = i_2)$$

□

Neben der eigenschaftsorientierten Sicht auf Systeme steht uns mit den Transitionssystemen eine ablauforientierte Sicht zur Verfügung. Auch für diese Sichtweise wollen wir nun Modifikationen einführen.

4.2.3 Modifikationen von Transitionssystemen

Ein Transitionssystem kann auf zwei natürliche Arten modifiziert werden, und zwar durch das Hinzufügen von Transitionen einerseits und durch das Entfernen von Transitionen andererseits. Die Modifikation des Zustandsraumes spielt eine nur untergeordnete Rolle, wie wir im folgenden noch diskutieren werden. Wir definieren die Modifikation eines Transitionssystems daher durch die Angabe entsprechender Transitionsmengen wie folgt:

Definition 4.5 *Modifikation von Transitionssystemen*

Sei $\mathcal{S} = (I, O, A, \Upsilon, T)$ *ein Transitionssystem, und seien* E *und* F *Mengen von Transitionen* $E, F \subseteq \mathit{VAL} \times \mathit{VAL}$*, wobei* F *die Forderung (d) aus Definition 3.5 erfüllt. Die Modifikation*

$$\mathcal{M} = (E, F)$$

entfernt die Menge E *der Transitionen aus der Transitionsmenge* T*, und fügt die Transitionen aus* F *hinzu:*

$$\mathcal{S} \triangle \mathcal{M} \stackrel{df}{=} (I, O, A, \Upsilon, (T \setminus E) \cup F)$$

□

Die Forderung an F beschränkt uns in der Modifikation von Transitionssystemen. So sollen auch modifizierte Systeme die Eigenschaft behalten, das Lesen von Nachrichten nicht mehr rückgängig machen zu können, nur bereits gesendete Nachrichten zu lesen

und einmal geschriebene Ausgaben nicht mehr löschen zu können. Die Beweisprinzipien aus Abschnitt 3.8 erfordern diese Eigenschaften. Wir werden in dieser Arbeit von nun an voraussetzen, dass alle Transitionen in diesem Sinne wohlgeformt sind. Präziser formuliert lautet die Transitionsmenge eines modifizierten Systems $(T \setminus E) \cup F'$, wobei $F' \subseteq F$ den Anteil von F beschreibt, der die notwendigen Bedingungen erfüllt.

Die für alle Systeme $\mathcal{S}$ neutrale Modifikation kann durch $(\varnothing, \varnothing)$ dargestellt werden, da mit ihr weder Transitionen ergänzt noch entfernt werden. Selbstverständlich stellt eine Modifikation (E, F) eine neutrale Modifikation dar, wenn $T \cap E = \varnothing$ und $F \subseteq T$ gilt, da in diesem Fall nur Transitionen entfernt werden, die gar nicht enthalten sind bzw. nur Transitionen ergänzt werden, die bereits in der Transitionsmenge liegen. Wir schließen derartige Fälle nicht von vornherein aus.

Durch Verwendung eines E mit $T \subseteq E$ wird die Transitionsmenge eines Systems auf ein beliebig wählbares F gesetzt, da damit alle Transitionen in T entfernt werden, und genau F verbleibt. Erneut läßt sich die Vervollständigung zu einem System mit chaotischem Verhalten durch Hinzunehmen aller Transitionen mittels $F = \mathit{VAL} \times \mathit{VAL}$ erreichen (natürlich beschränkt auf die Transitionen, die die Forderung (d) aus Definition 3.5 erfüllen).

Ein Transitionssystem enthält noch die Variablenmenge A und die Initialbedingung Υ als weitere Bestandteile, die sich modifizieren ließen. Änderungen dieser Anteile haben wir jedoch nicht in $\mathcal{M}$ aufgenommen, da sie auf Modifikationen der Transitionen reduziert werden können:

- Ein Transitionssystem kann gemäß dem in Abschnitt 3.6 beschriebenen Systemmodell beliebige Variablen aus der Menge der Variablen VAR verwenden. Die Variablenmenge A dient dazu, die in einem System vorkommenden Variablen zu dokumentieren, und damit die Kompositionalität leichter definieren zu können. Es ist damit durch reine Inspektion von I, O und A möglich zu erkennen, welche Variablen gelesen und welche geschrieben werden dürfen. Solange sichergestellt ist, dass bei der Komposition nicht etwa auf die interne Variablen einer anderen Komponente zugegriffen wird, dürfen ohne weiteres neue Variablen in A ergänzt werden. Da es beliebig viele „frische“ Variablen gibt, können Konflikte leicht vermieden werden.

  Allerdings ist zu beachten, dass eine neue ergänzte Variable in den existierenden Transitionen bislang nicht vorkommt, und daher von dieser in beliebiger Weise bei jedem Systemschritt verändert werden kann. Dies kann bei Bedarf mit Hilfe einer Modifikation durch Entfernen von Transitionen leicht behoben werden. Wird eine neue Variable $v \in \mathit{VAR}$ in A aufgenommen, so werden durch eine Modifikation $\mathcal{M} = (E, F)$ mit

  $$E \supseteq \{(\alpha, \beta) \mid \alpha(v) \neq \beta(v)\}$$

  Veränderungen von v ausgeschlossen, die erst durch explizite Aufnahme durch entsprechende Transitionen via F wieder eingeführt werden können. Dieser Ansatz wird im Beispiel 4.5 veranschaulicht.

Die Entfernung von Variablen aus A kann selbstverständlich zu Problemen führen, wenn die zu entfernende Variable in den Transitionen verwendet wird; eine Systemspezifikation wäre dann nicht mehr konsistent. Wir lassen die Entfernung von Variablen aus A daher nicht zu. Soll ein System auf eine Art und Weise modifiziert werden, die eine Variable überflüssig macht, so kann durch entsprechende Modifikationen der Transitionen diese Variable einfach nicht mehr „benutzt", also auf irgendeine Art lesend oder schreibend referenziert werden.

- Die Initialbedingung Υ zeichnet eine Menge von Startzuständen aus, die diese Bedingung erfüllen. Eine Modifikation der Initialbedingung ließe sich wiederum durch eine Kombination von Verstärkung und Abschwächung beschreiben, verbunden mit einer Verkleinerung bzw. Vergrößerung der entsprechenden Zustandsmenge. Zur Vereinfachung des Formalismus wollen wir dies aber wie folgt reduzieren auf eine Modifikation der Transitionsmengen.

  Zunächst wandeln wir dazu ein System $\mathcal{S}$ um, ohne sein Verhalten dabei zu ändern: Wir führen – wie bereits beschrieben – eine „frische" boolesche Variable *init* ein, die von keiner existierenden Transition verändert werden darf. Dazu sei E entsprechend definiert als die Menge, die alle Transitionen enthält, die *init* verändern. Den neuen Startzustand charakterisieren wir als $init = \mathsf{true}$. Wir fügen nun neue Transitionen hinzu, die vom neuen Startzustand zu den bisherigen Startzuständen führen:

  $$F = \{(\alpha, \beta) \mid \alpha \models init \ \wedge \ \beta \models (\Upsilon \wedge \neg\, init)\}$$

  Das resultierende System

  $$\mathcal{S} = (I, O, A \cup \{init\}, init = \mathsf{true}, (T \setminus E) \cup F)$$

  hat demnach das gleiche Verhalten wie $\mathcal{S}$ (die Modifikation (E, F) stellt in diesem konkreten Fall also eine neutrale Modifikation dar), aber mit einer vereinfachten, minimalen Initialbedingung.

  Soll das System nun in seinen Startzuständen modifiziert werden, so ist dies mit reinen Modifikationen entsprechend Definition 4.5 möglich.

Für eine praktische Beschreibung von Modifikationen ist selbstverständlich eine Beschreibungstechnik für Modifikationen von A und Υ wünschenswert, die auf die beschriebenen schematischen Umsetzungen verzichtet. Da in dieser Arbeit aber prinzipielle Überlegungen im Vordergrund stehen, wollen wir hier darauf aber verzichten.

Beispiel 4.5 Wir modellieren zunächst die Entscheidung, von welchem Kanal die modifizierte Komponente *Merge*$\triangle\mathcal{M}$ ausschließlich liest, durch Verwendung einer neuen Variable c (*choice*), die unabhängig vom Startwert nichtdeterministisch auf einen der Werte 1 oder 2 gesetzt wird, der dann nicht mehr verändert wird. Die entsprechenden (zu ergänzenden) Transitionen τ_3 und τ_4 beschreiben wir tabellarisch:

$F \stackrel{df}{=}$

Name	*Pre*	*Input*	*Output*	*Post*
τ_3	$c \neq 1 \wedge c \neq 2$	–	–	$c = 1$
τ_4	$c \neq 1 \wedge c \neq 2$	–	–	$c = 2$

Da die Variable c noch nicht in A enthalten war, können die Transitionen τ_1 und τ_2 in T den Wert von c beliebig modifizieren. Dies schließen wir auf systematische Weise aus, indem wir alle Transitionen entfernen, die den Wert von c verändern, und definieren

$$E_1 \stackrel{df}{=} \{(\alpha, \beta) \mid \alpha(c) \neq \beta(c)\}$$

Nun sind noch die eigentlichen Transitionen so einzuschränken, dass tatsächlich nur von dem Kanal gelesen wird, der durch c angezeigt wird. So soll die Transition τ_1 nur schaltbereit sein, wenn $c = 1$ gilt. Diese Verstärkung der Vorbedingung spiegelt sich formal wider in der Entfernung aller Transitionen, die diese Vorbedingung *nicht* erfüllen. Also erreichen wir durch die Entfernung der Transitionen

$E_2 \stackrel{df}{=}$

Name	*Pre*	*Input*	*Output*	*Post*
τ_1'	$c \neq 1$	$i_1?a$	$o!a$	–
τ_2'	$c \neq 2$	$i_2?a$	$o!a$	–

den gewünschten Effekt. Unsere gesuchte Modifikation lautet also

$$\mathcal{M} \equiv (E_1 \cup E_2, F)$$

Die Negativformulierungen der entfernten Transitionen sind nicht sehr intuitiv. Errechnet man die resultierende Transitionsmenge des Systems $Merge \triangle \mathcal{M}$, so erhält man

Name	*Pre*	*Input*	*Output*	*Post*
τ_1''	$c = 1$	$i_1?a$	$o!a$	$c' = c$
τ_2''	$c = 2$	$i_2?a$	$o!a$	$c' = c$
τ_3	$c \neq 1 \wedge c \neq 2$	–	–	$c' = 1$
τ_4	$c \neq 1 \wedge c \neq 2$	–	–	$c' = 2$

□

Die Modifikationen, die wir bislang vorgestellt haben, nehmen unmittelbar Bezug auf die Beschreibungstechnik, in der die Systeme spezifiziert werden: Die relationalen Beschreibungen werden durch eine Modifikation der Relationen verändert, Black-Box Eigenschaften durch eine Veränderung der Eigenschaften und Transitionssysteme durch eine Manipulation der Transitionen. Eine Alternative stellt die Komposition mit Modifikationskomponenten dar, die wir im folgenden vorstellen werden.

4.2.4 Modifikationskomponenten

Modifikationskomponenten – vorgestellt in [8] – erlauben es uns, ein zu modifizierendes System unangetastet zu lassen, und die Veränderung in seinem Verhalten durch die *Komposition* mit einer weiteren Komponente zu beschreiben. Die Modifikationskomponente wird zwischen die Umgebung und die eigentliche Komponente geschaltet, und kann die ein- und ausgehenden Nachrichten verändern, löschen oder sogar neue Nachrichten erzeugen. Sowohl die Modifikationskomponente als auch die zu modifizierende Komponente können dabei unabhängig voneinander in beliebig wählbaren Formalismen beschrieben werden.

Verschiedene Varianten, die sich in Ausdrucksmächtigkeit und Komplexität unterscheiden, sind in Abbildung 4.3 graphisch dargestellt. Mit Hilfe einer *vorgeschalteten* Modifikationskomponente entsprechend Abbildung 4.3(a) können eingehende Nachrichten

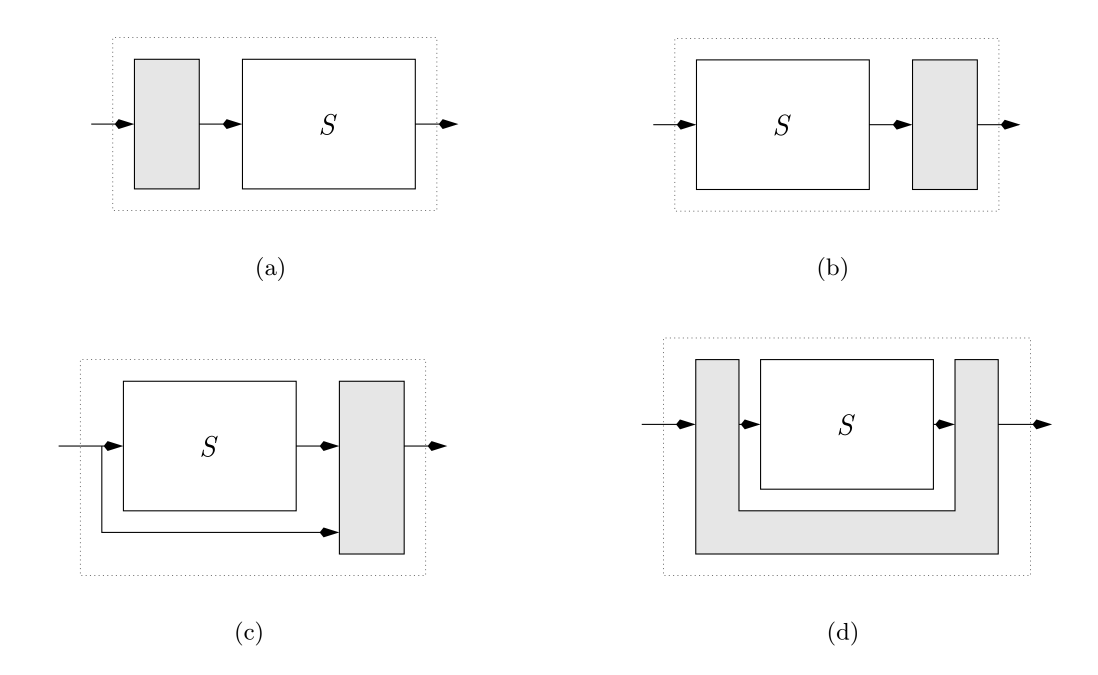


Abbildung 4.3: Varianten von Modifikationskomponenten

manipuliert werden. Der Verlust von Eingabenachrichten ist beispielsweise auf diese Weise modellierbar.

Eine Modifikation der Ausgabe ist mit der in Abbildung 4.3(b) dargestellten Variante möglich. Der Verlust oder die Verfälschung von Ausgaben kann damit gut modelliert werden.

Soll die Modifikation der Ausgaben auch von der Eingabe abhängen, so ist nicht in jedem Fall ausreichend Information an den Ausgabekanälen verfügbar. Variante 4.3(c) schafft hier Abhilfe, da die Modifikationskomponente auch Kenntnis über die Eingabe erhält. Im Prinzip ist diese Variante bereits mächtig genug, um Modifikationen hin zu einem beliebigem Verhalten zu modellieren: $\mathcal{M}$ könnte die Ausgaben von S ignorieren, und ein beliebiges, wählbares Verhalten erbringen. Wünschenswert ist aber vielmehr, die Funktionalität von S zu nutzen und in $\mathcal{M}$ wirklich nur die eigentliche Abweichung vom Normalverhalten zu modellieren.

Die in Abbildung 4.3(d) dargestellte Variante bietet hier noch mehr Ausdrucksmächtigkeit, da sowohl die Ein- als auch Ausgabekanäle modifiziert werden können. Die Modifikation kann unter Ausnutzung aller Informationen definiert werden, die in einer Black-Box Sicht verfügbar sind, d.h. mit Kenntnis der gesamten bisherigen Ein- und Ausgabegeschichten der Komponente S.

Die vier erwähnten Varianten von Modifikationskomponenten werden wir nun formal definieren.

Definition 4.6 *Modifikationskomponenten*

Sei S ein System mit der Schnittstelle (I, O). Eine Komponente mit der Schnittstelle (X, Y) heißt Modifikationskomponente *zu S, wenn eine der folgenden Bedingungen erfüllt ist:*

1. $X \Bumpeq I \wedge Y = I$
   Die Eingabe von $\mathcal{M}$ ist kompatibel zur Eingabe von I (ändert also nicht die Schnittstelle, nur die Namen der Eingabekanäle) und $\mathcal{M}$ kann vor S geschaltet werden (Abbildung 4.3(a)).

2. $X = O \wedge Y \Bumpeq O$
   Die Modifikationskomponente kann direkt hinter S geschaltet werden, und ihre Ausgabe ändert die Schnittstelle nicht bis auf Umbenennung der Kanäle (Abbildung 4.3(b)).

3. $X = (I \cup O) \wedge Y \Bumpeq O$
   Die Eingabe der Komponente $\mathcal{M}$ wird sowohl mit der Eingabe als auch mit der Ausgabe von S verbunden, während die Ausgabe von $\mathcal{M}$ zur Ausgabe von S kompatibel bleibt (Abbildung 4.3(c))

4. $\exists\, I', O' \bullet X = (I' \cup O) \wedge Y = (I \cup O') \wedge I' \Bumpeq I \wedge O' \Bumpeq O$
   Die Modifikationskomponente empfängt sowohl Eingaben von der Umgebung über I' als auch die Ausgaben von S über O. Sie kann Ausgaben machen über O' an die Umgebung und über I an S (Abbildung 4.3(d)).

Die Modifikation wird definiert durch die Komposition mit der entsprechenden Modifikationskomponente. Das modifizierte System wird also ausgedrückt durch

$$S \otimes \mathcal{M}$$

Es ergibt sich in allen vier Fällen, dass das komponierte System $S \otimes \mathcal{M}$ kompatibel zu S ist, also vermöge einer Umbenennung der Kanäle in jedem Kontext an Stelle von S eingesetzt werden kann. □

Durch die Beschränkung auf eine Modifikation des Black-Box Verhaltens sind Modifikationskomponenten nicht immer gut geeignet zur Repräsentation von Fehlern. So läßt sich beispielsweise eine spontane, interne Veränderung im Datenanteil einer Komponente nur kompliziert darstellen. Soll eine Veränderung des Inhalts einer Speicherzelle durch eine Modifikationskomponente modelliert werden, so müssten alle ausgehenden Nachrichten, die aus lesenden Zugriffen auf diese Speicherzelle resultieren, konsequent modifiziert werden, bis in diese Speicherzelle schließlich neu geschrieben wird. Diese Modifikation ist durch eine entsprechende zusätzliche Transition deutlich einfacher zu repräsentieren.

Beispiel 4.6 Die Modifikation von *Merge* zu einer Komponente, die nur einen Eingabekanal überträgt, läßt sich auf verschiedene Weisen modellieren, von denen wir hier zwei skizzieren:

- Eine vorgeschaltete Modifikationskomponente $\mathcal{M}$ vom Typ entsprechend Abbildung 4.3(a) kopiert nur einen der beiden Eingabeströme, und blockiert den jeweils anderen. Als Schnittstelle für $\mathcal{M}$ wählen wir $(\{i'_1, i'_2\}, \{i_1, i_2\})$, womit Bedingung 1 erfüllt ist mittels der Bijektion $f = \{i'_1 \mapsto i_1, i'_2 \mapsto i_2\}$.

  Das Verhalten von $\mathcal{M}$ beschreiben wir mit Hilfe einer Black-Box Spezifikation durch

  $$\Psi \equiv (i_1 = i'_1 \wedge i_2 = \langle\rangle) \vee (i_1 = \langle\rangle \wedge i_2 = i'_2)$$

  Das modifizierte System hat die Schnittstelle $(\{i'_1, i'_2)\}, \{o\})$, und zeigt das durch Konjunktion beschriebene Verhalten

  $$\Psi \wedge D_1 \circledS o = i_1 \wedge D_2 \circledS o = i_2$$

  woraus sich durch Expansion und Ersetzung

  $$(D_1 \circledS o = i'_1 \wedge D_2 \circledS o = \langle\rangle) \vee (D_1 \circledS o = \langle\rangle \wedge D_2 \circledS o = i'_2)$$

  ableiten läßt. Nach einer Vereinfachung ergibt sich (bis auf Kanalumbenennung) genau die in Beispiel 4.4 erwartete Eigenschaft:

  $$o = i'_1 \vee o = i'_2$$

- Der Verlust der Daten eines ganzen Eingabekanals kann auch durch eine nachgeschaltete Modifikationskomponente (entsprechend Abbildung 4.3(b)) modelliert werden. Wir wählen die Komponente $\mathcal{M}$ mit der Schnittstelle $(\{o\}, \{o'\})$. Das Verhalten kann wieder leicht als Black-Box Spezifikation durch

  $$(o' = D_1 \circledS o) \quad \vee \quad (o' = D_2 \circledS o)$$

  angegeben werden. Selbstverständlich ist es auch möglich, das Verhalten von $\mathcal{M}$ in anderen Beschreibungstechniken anzugeben. Exemplarisch spezifizieren wir das Verhalten durch ein Transitionssystem

  $$\mathcal{S} = (\{o\}, \{o'\}, \{o^\circ, c\}, c = 1 \vee c = 2, T)$$

  mit einem tabellarisch definierten T:

$T =$

Name	*Pre*	*Input*	*Output*	*Post*
τ_1	$c = 1 \wedge d \in D_1$	$o?d$	$o'!d$	$c' = c$
$\tau_{1'}$	$c = 1 \wedge d \in D_2$	$o?d$	$-$	$c' = c$
$\tau_{2'}$	$c = 2 \wedge d \in D_1$	$o?d$	$-$	$c' = c$
τ_2	$c = 2 \wedge d \in D_2$	$o?d$	$o'!d$	$c' = c$

□

4.3 Formalisierung der Fehlerbegriffe

In der Literatur werden die Begriffe Fehler (engl. *fault*), Fehlzustand (engl. *error*) und Ausfall bzw. Versagen (engl. *failure*) meist nur informell beschrieben. Sogar in Standardwerken wie beispielsweise [43, 44, 66] wird auf eine präzise Definition verzichtet, da eine formale Grundlage nicht zur Verfügung steht. Mit dem in 4.2 vorgestellten Begriff der Modifikationen können wir nun geeignete formale Definitionen für *Fehler*, *Fehlerzustand* und *Versagen* vorzuschlagen.

4.3.1 Fehler

Wie in Kapitel 2.1 bereits diskutiert, bezeichnet man einen Fehler üblicherweise als den eigentlichen, ursprünglichen Grund für die Abweichung eines konkreten, fehlerbehafteten Systems von seiner korrekten, intendierten Fassung. Ein Fehler ist also genau der Defekt, der ein System zu einem fehlerhaften macht.

Ein Fehler muss nicht unbedingt einfach zu lokalisieren sein, wie zum Beispiel ein Tippfehler, ein falsches Datum oder ein verfehlter Prozeduraufruf in einem Programmcode. Ein Entwurfsfehler, inkompatible Schnittstellen oder inkonsistente Eigenschaften eines Systems können zu einer informellen Bewertung wie *„Das System hat einen Fehler"* führen, ohne dass man auf den Fehler unmittelbar „zeigen" kann.

Der Begriff der Modifikation wurde unter anderem dadurch motiviert, derartige Abweichungen zu repräsentieren. Wir definieren Fehler daher direkt als Modifikationen:

Definition 4.7 *Fehler*

Ein Fehler *eines fehlerbehafteten Systems $S \triangle \mathcal{M}$ bezüglich einer als korrekt definierten Spezifikation S wird beschrieben durch die Modifikation $\mathcal{M}$.* □

Diese Definition charakterisiert den Fehler eines Systems $S \triangle \mathcal{M}$, also für ein System, das den Fehler bereits explizit in seiner Beschreibung enthält. Für ein fehlerbehaftetes System S', das relativ zu einer Spezifikation S einen Fehler aufweist, muss dieser Fehler erst gefunden werden. Der Fehler in S' ist dann das (nicht notwendigerweise eindeutige) $\mathcal{M}$, für das

$$S' = S \triangle \mathcal{M}$$

gilt. Die Beschreibungstechnik spielt in Definition 4.7 keine Rolle. Fehler können also insbesondere mit Black-Box Spezifikationen, Transitionssystemen und Modifikationskomponenten beschrieben werden. Es sei bemerkt, dass wir nicht unterscheiden zwischen *einem* und *mehreren* Fehlern in einem System, da Fehler unter Umständen nicht eindeutig lokalisiert und damit auch nicht sinnvoll quantifiziert werden können.

4.3.2 Fehlzustand

Weist ein System einen Fehler auf, so kann dieser in einem Ablauf eines betroffenen Systems eine Wirkung haben, und zwar sowohl auf interne Zustände als auch auf das nach aussen sichtbare Verhalten. Nach aussen sichtbares Fehlverhalten werden wir noch unter dem Begriff *Versagen* diskutieren. Die Sichtbarkeit ist in unserem Systemmodell durch die Festlegung der Schnittstelle jedes Systems gegeben: Die in (I, O) enthaltenen Kanäle sind sichtbar, während andere Bestandteile systemintern und für die Umgebung unsichtbar bleiben.

Die Beschreibungstechnik der Black-Box Sichtweise wie auch die Modifikationskomponenten nehmen keinerlei Bezug zu Interna von Systemen. Daher können mit ihnen keine internen Zustände eines Systems und somit auch keine Fehlzustände charakterisiert werden. Mit dem Formalismus der Transitionssysteme steht uns aber eine weitere Sicht auf

Systeme zur Verfügung, mit dem Zustände und damit auch Fehlzustände angemessen dargestellt werden können. Wir definieren Fehlzustände als die Zustände in einem Systemablauf, die nur unter Verwendung von Fehlertransitionen (aus F mit $\mathcal{M} = (E, F)$) erreicht werden können:

Definition 4.8 *Fehlzustand*

Sei ein Transitionssystem $\mathcal{S}$ und eine Modifikation $\mathcal{M} = (E, F)$ gegeben. Die Menge der Fehlzustände $ERROR(\mathcal{S}, \mathcal{M})$ enthält alle Zustände α, die in einem Ablauf des modifizierten Systems erreicht werden können, aber nur unter Verwendung von Fehlertransitionen aus F:

$$\begin{aligned} ERROR(\mathcal{S},\mathcal{M}) \stackrel{df}{=} \{\alpha \mid \ & \exists\, \xi \in \langle\!\langle \mathcal{S} \triangle \mathcal{M} \rangle\!\rangle \bullet \exists\, k \in \mathbb{N} \bullet \xi.k = \alpha \quad \wedge \\ & \forall\, \xi \in \langle\!\langle \mathcal{S} \triangle \mathcal{M} \rangle\!\rangle, k \in \mathbb{N} \bullet \\ & \xi.k = \alpha \ \Rightarrow \ \exists\, l < k \bullet (\xi.l, \xi.(l+1)) \in F\} \end{aligned}$$ □

Die Menge E von Transitionen, die im modifizierten System nicht mehr ausgeführt werden können, reduziert im allgemeinen die Menge der erreichbaren Zustände und damit auch die Zahl der Fehlzustände. Eine alternative Definition in [9] verzichtet auf die Forderung der Erreichbarkeit und unterscheidet daher nicht zwischen Zuständen, die durch die Modifikation unerreichbar werden und Zuständen, die bereits ohne Modifikation unerreichbar waren. Für die hier gewählte Definition sprechen zwei Argumente:

- Der vorgestellte Formalismus dieser Arbeit zielt darauf ab, mit Fehlern *explizit* umgehen zu können, indem die Fehler, ihre Auswirkungen und Gegenmaßnahmen modelliert werden. Wie in Abbildung 4.1 skizziert, soll damit eine klare Grenze gezogen werden zwischen einerseits den Fehlern, mit denen man rechnet umgehen möchte und andererseits den Fehlern, für die man eine Behandlung nicht vorsieht. Obige Definition unterstützt diese klare Trennung.

- In Abschnitt 5.3 werden Kriterien angegeben, die ein Prädikat zur Erkennung von Fehlern zu erfüllen hat. Durch die Aufnahme unerreichbarer Fehler in die Menge der Fehlerzustände müßten auch diese (unnötigerweise) berücksichtigt werden.

Nicht jeder Zustand α, der über fehlerhafte Transitionen erreicht werden kann, ist unbedingt ein Fehlzustand. Ist nämlich auch ein anderer Ablauf möglich, der zu demselben Zustand α führt, ohne dabei Fehlertransitionen zu verwenden, so ist α nicht in der Menge $ERROR(\mathcal{S}, \mathcal{M})$ enthalten. Abbildung 4.4 veranschaulicht einen derartigen Fall. Es werden zwei Abläufe mit gemeinsamen Zuständen dargestellt. Runde Knoten repräsentieren Zustände, die keine Fehlzustände sind; quadratische Knoten sind Fehlerzustände; die fett gezeichneten Pfeile repräsentieren Transitionen aus F. Der mit x markierte Knoten ist *kein* Fehlzustand, da dieser auch über einen fehlerfreien Ablauf erreichbar ist.

Wären alle Nachfolgerknoten von Fehlzuständen auch wieder Fehlzustände, wären Mechanismen zur Fehlerkorrektur nicht sinnvoll definierbar.

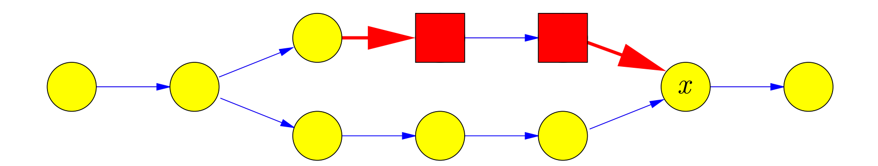


Abbildung 4.4: Fehlerzustände in einem Ablauf

4.3.3 Versagen

Ein *Versagen* (engl. *failure*) wird üblicherweise als eine (von aussen) sichtbare Abweichung des Verhaltens vom spezifizierten Verhalten bezeichnet. Wir stellen diesen Begriff wieder im Rahmen der Black-Box Spezifikationen und Transitionssysteme dar.

Bei Black-Box Sichten definieren wir Versagen auf naheliegende Weise als die Menge der Verhalten, die im modifizierten System auftreten können, im spezifizierten Verhalten aber nicht. Eine derartige Abweichung vom spezifizierten Verhalten ist unmittelbar für die Umgebung sichtbar, da sie in der Black-Box Sicht formuliert wird.

Für die Transitionssysteme ist ein Ablauf, der nicht dem spezifizierten System entspricht, nicht unbedingt als Versagen zu bezeichnen, da eine Abweichung auch lediglich *intern* auftreten kann und damit „nur“ einen Fehlzustand darstellt. Wir definieren das Versagen eines Systems daher auf folgende Weise: Ein Versagen tritt in einem Zustand auf, wenn alle Zustände, die von außen betrachtet identisch mit ihm sind, Fehlzustände sind. Dies bedeutet, dass bei einem Versagen das nach aussen sichtbare Verhalten eines Transitionssystems nur aufgrund fehlerhafter Transitionen zustande kommen konnte.

Definition 4.9 *Versagen*

1. *Seien eine Black-Box Spezifikation S und eine Modifikation $\mathcal{M}$ gegeben. Die Menge aller Verhalten, die ein Systemversagen aufweisen, wird definiert durch*

$$FAILURES(S, \mathcal{M}) \stackrel{df}{=} \{(i, o) \mid (i, o) \in [\![S \triangle \mathcal{M}]\!] \wedge (i, o) \notin [\![S]\!]\}$$

2. *Für ein Transitionssystem $\mathcal{S}$ und eine zugehörige Modifikation $\mathcal{M}$ wird die Menge der Zustände, die ein Systemversagen beschreiben, definiert durch*

$$FAILURES(\mathcal{S}, \mathcal{M}) \stackrel{df}{=} \{\alpha \mid \forall \beta \bullet \beta \stackrel{I \cup O}{=} \alpha \Rightarrow \beta \in ERROR(\mathcal{S}, \mathcal{M})\}$$

□

Das Versagen von Systemen läßt sich im Rahmen der Black-Box Sicht weiter vereinfachen. Aufgrund der Tautologie

$$((\Phi \wedge \Phi_E) \vee \Phi_F) \wedge \neg \Phi \quad \Leftrightarrow \quad \Phi_F \wedge \neg \Phi$$

können wir die Menge der Versagen auch folgerndermaßen charakterisieren:

Proposition 4.4 *Black-Box Charakterisierung von Versagen*

Ist ein System S und seine Modifikation durch eine Black-Box Spezifikation Φ und $\mathcal{M} = (\Phi_E, \Phi_F)$ beschrieben, so gilt

$$FAILURES(S, \mathcal{M}) = \{(i, o) \mid \Phi_F \wedge \neg \Phi\}$$

□

Die Abschwächung Φ_E hat demzufolge keinen Einfluß auf die Menge der Versagen eines Systems. Da Φ_E nur zusätzliche Eigenschaften beschreibt, die das modifizierte System zu erfüllen hat, können dadurch natürlich keine Abweichungen vom normalen Systemverhalten bewirkt werden. Nur Φ_F gibt dem System neue Freiheiten, die zu sichtbaren Abweichungen führen können. In vielen formalen Ansätzen zur Fehlermodellierung wird daher auf die Modellierung von Eigenschaftsverstärkungen daher auch verzichtet. Unser Begriff der *Modifikationen* ist somit allgemeiner.

Auf eine Unterscheidung der Begriffe *Ausfall* und *Versagen*, wie sie auch in Kapitel 2 angesprochen wird, haben wir verzichtet, da dies in unserem Kontext nicht sinnvoll erscheint. Das Ziel der Arbeit ist es, die Beschreibung von Systemen und ihren Modifikationen zu ermöglichen. Ob der Ausfall physikalischer Natur ist oder nicht, spielt hierbei keine prinzipielle Rolle, solange wir in der Lage sind, die Auswirkungen zu modellieren.

Beispiel 4.7 Um den Unterschied zwischen einem Fehlzustand und einem Versagen veranschaulichen zu können, definieren wir einen *Zähler*, der einen internen Zustand besitzt. Der Zähler habe zwei Eingabekanäle: Auf i empfängt er zu zählende Daten vom Typ $D_1 \cup D_2$, auf r kann er Anfragen R empfangen, die ihn auffordern, auf dem Ausgabekanal c die bisherige Anzahl von gezählten Daten auszugeben. Formal haben wir also die Schnittstelle

$$I = \{i, r\}, O = \{c\} \text{ mit } type(i) = D_1 \cup D_2, type(r) = \{R\}, type(c) = \mathbb{N}$$

und definieren

$$Z\ddot{a}hler = (I, O, \{i^\circ, r^\circ, n\}, n = 0, T)$$

mit $n \in \mathbb{N}$ und T definiert durch

Name	*Pre*	*Input*	*Output*	*Post*
τ_1	–	$r?R$	$c!n$	$n' = n$
τ_2	–	$i?d$	–	$n' = n + 1$

Wir definieren eine Modifikation $\mathcal{M}$, die spontan Daten vom Eingabekanal i verliert, also Daten unter Umständen nicht mitzählt: $\mathcal{M} \overset{df}{=} (\varnothing, F\}$ mit

$F \overset{df}{=}$

Name	*Pre*	*Input*	*Output*	*Post*
τ_3	–	$i?d$	–	$n' = n$

Betrachten wir folgenden exemplarischen Ablauf, so erkennen wir, dass Zustand 4 durch die fehlerhafte Transition τ_3 erreicht wurde: Es wurde ein Datum von o gelesen, aber n nicht erhöht:

	n	i°	r°	c
1	0	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$
2	1	$\langle d_1\rangle$	$\langle\rangle$	$\langle\rangle$
3	1	$\langle d_1\rangle$	$\langle R\rangle$	$\langle 1\rangle$
4	1	$\langle d_1, d_2\rangle$	$\langle R\rangle$	$\langle 1\rangle$
5	2	$\langle d_1, d_2, d_3\rangle$	$\langle R\rangle$	$\langle 1\rangle$
6	2	$\langle d_1, d_2, d_3\rangle$	$\langle R, R\rangle$	$\langle 1, 2\rangle$

Dieser Zustand ist ein *Fehlzustand*, da es keinen Ablauf ohne Verwendung von τ_3 gibt, der diesen Zustand erreicht: Im unmodifizierten System *Zähler* ist $n = \#i^\circ$ eine Invariante.

Dennoch beschreibt Zustand 4 kein *Versagen* des Systems, d.h. der Fehler ist noch nicht sichtbar geworden, da es einen fehlerfreien Ablauf gibt, der nach aussen identisch aussieht. Zustand 4′ unterscheidet sich von Zustand 4 durch den korrekten Wert für n, aber n ist als lokale Variable vor der Umgebung verborgen:

	n	i°	r°	c
1	0	$\langle\rangle$	$\langle\rangle$	$\langle\rangle$
2	1	$\langle d_1\rangle$	$\langle\rangle$	$\langle\rangle$
3	1	$\langle d_1\rangle$	$\langle R\rangle$	$\langle 1\rangle$
4′	2	$\langle d_1, d_2\rangle$	$\langle R\rangle$	$\langle 1\rangle$

Auch Zustand 5 ist ein Fehlzustand, der aber noch unsichtbar bleibt. Erst im Zustand 6 tritt ein Versagen auf, da durch die zweite Anfrage R der interne Fehler offensichtlich wird: Das System $Z\ddot{a}hler \triangle \mathcal{M}$ reagiert mit einer 2, während das fehlerfreie System eine 3 ausgegeben hätte.

Im Falle einer *Black-Box Spezifikation* sieht dieser Fall gänzlich anders aus. Da wir uns auf ein ungezeitetes Systemmodell beschränkt haben, kann der Zähler nur sehr grob spezifiziert werden durch

$$\begin{aligned}
&\#c = \#r \quad \wedge\\
&j < k \Rightarrow c.j \leq j.k \quad \wedge\\
&max\{n \mid \exists\, k \bullet c.k = n\} \leq \#i
\end{aligned}$$

Auf jede Anfrage wird also reagiert, die ausgegebenen Zahlen wachsen monoton, und es können nie mehr Daten gezählt werden als auf i verfügbar sind. Es ist ohne explizite Modellierung der Zeit nicht erkennbar, in welcher Reihenfolge die zu zählenden Daten und Anfragen vom System empfangen wurden. So ist es möglich, dass ein Datum, dann ein R, wieder ein Datum, das zweite R und schließlich das dritte Datum empfangen wurden. Die Ausgabe $\langle 1, 2\rangle$ wäre also eine korrekte und spezifizierte Reaktion des Systems. Insgesamt sind im ungezeiteten Modell auf die Eingabe $i = \langle d_1, d_2, d_3\rangle$ und $r = \langle R, R\rangle$ folgende Reaktionen möglich:

$$c \in \{\langle 0,0\rangle, \langle 0,1\rangle, \langle 0,2\rangle, \langle 0,3\rangle, \langle 1,1\rangle, \langle 1,2\rangle, \langle 1,3\rangle, \langle 2,2\rangle, \langle 2,3\rangle, \langle 3,3\rangle\}$$

Der Verlust eines Datums macht sich im Verhalten dieses Beispiels nur dadurch bemerkbar, dass die Zahl 3 nicht mehr ausgegeben werden kann. Die Modifikation wird also durch eine Einschränkung mittels Φ_E ausgedrückt, und es liegt in unserem Modell kein *Versagen* im Sinne von Definition 4.9 vor. Die auf dem Transitionssystem basierende Spezifikation erlaubt also eine deutlich differenziertere Untersuchung. □

In modifizierten Transitionssystemen lassen sich neben der Menge der normalen Zustände (die auch im unmodifizierten System erreicht werden können), der Menge der Fehlzustände und der Menge der Versagenszustände noch weitere Klassen von Zuständen auszeichnen:

- *Zustände, die zu Fangzuständen geworden sind*, da die von ihnen ausgehenden Transitionen durch E entfernt wurden. Das System wäre in solchen Zuständen nicht mehr reaktiv und würde seine Aktivität beenden. Können diese Zustände durch einen dem System neu ergänzten Mechanismus erkannt werden, so kann man auch eine entsprechende Reaktion in das System integrieren, wie wir in Kapitel 5 diskutieren.

- *Zustände, die aufgrund entfernter Transitionen nicht mehr erreichbar sind.* Können Lebendigkeitseigenschaften mit dem Erreichen bestimmter Zustände verknüpft werden, so kann man untersuchen, ob eine Modifikation die Erreichbarkeit der Knoten und damit eine Systemeigenschaft gefährdet. Wir werden in Abschnitt 5.7.2 zeigen, wie man die Auswirkungen von entfernten Transitionen im Rahmen von Beweisdiagrammen ermittelt.

Die in diesem Abschnitt definierten Begriffe können zur Definition von Techniken zur Fehlererkennung und -korrektur verwendet werden und werden im entsprechenden Abschnitt 5.3 wieder aufgegriffen.

4.4 Fehlertoleranz

Fehlertoleranz läßt sich auf verschiedene Weisen interpretieren. Meist wird sie als die Eigenschaft eines Systems betrachtet, auf auftretende Fehlern auf eine noch akzeptable Weise zu reagieren. Eine präzise Fehlertoleranzaussage macht diese Aussage explizit. In einer alternativen Interpretation des Begriffes läßt sich Fehlertoleranz als die Fähigkeit eines Systems verstehen, die Wirkung von auftretenden Fehlern *einzudämmen*. Wir werden beide Interpretationen im folgenden diskutieren.

4.4.1 Fehlertoleranz als korrespondierende Modifikation

Betrachtet man *Fehlertoleranz* als die Eigenschaft eines Systems, auf das Auftreten von Fehlern auf eine akzeptable Weise zu reagieren, müssen die folgenden zwei Fragen beantwortet werden, um diese Aussage explizit zu machen:

- *Welche Fehler werden toleriert?*

  Ein System kann im allgemeinen nicht beliebige Fehler tolerieren, sondern nur eine bestimmte Fehlerklasse. In der Literatur finden sich teilweise standardisierte Fehlermodelle, auf die sich Aussagen zur Fehlertoleranz beschränken, beispielsweise in [60]. Eine größere Zahl wird aufgeführt in [23], darunter die Fehlerklassen *ommision*, *timing*, *response*, *crash failures* und mehr. Präzisere Aussagen über Fehlertoleranz lassen sich machen, wenn die Fehlerklassen individuell für gewünschte Aussagen der Fehlertoleranz formuliert werden können, wie dies beispielsweise in den Arbeiten [5, 36, 47] durchgeführt wurde: Fehler werden darin durch eine beliebig definierbare Menge zusätzlicher Transitionen modelliert.

- *Welches Verhalten des Systems wird noch als tolerierbar akzeptiert?*

  Im allgemeinen verändern Fehler das Systemverhalten, so dass ein fehlerbehaftetes System in der Regel nicht mehr alle Forderungen einer Spezifikation erfüllt. Eine präzise Aussage zur Fehlertoleranz muss definieren, inwieweit sich ein Systemverhalten ändern darf, um dennoch als akzeptabel zu gelten. In der Literatur finden sich dazu Begriffe wie beispielsweise *Toleranzspezifikation* [5] und *graceful degradation* (sanfte Leistungsminderung [31]).

Die in einem System vorhandenen Fehler können wir als Modifikationen darstellen und ebenso die Verletzung von Umgebungsannahmen durch eine Abschwächung der sie beschreibenden Aussage formulieren. Damit werden die Fehler, die toleriert werden sollen, ausgezeichnet. Das tolerierbare Verhalten, das ein System trotz Fehler aufweisen soll, können wir relativ zum Normalverhalten ebenfalls durch eine Modifikation beschreiben.

Damit sind wir mit unserem Systemmodell also in der Lage, präzise Antworten auf die beiden obigen Fragen geben zu können. Dies ermöglicht es uns, eine Aussage über Fehlertoleranz eines Systems formal zu beschreiben und – wenn sie von einem System erfüllt wird – auch zu verifizieren.

Definition 4.10 *Fehlertoleranz*

Sei ein Transitionssystem $\mathcal{S}$ gegeben, das die Eigenschaft Φ habe, d.h. es gelte

$$\mathcal{S} \models \Phi$$

Wir nennen das System $\mathcal{S}$ fehlertolerant *bezüglich der Fehler $\mathcal{M}^S$ und der Toleranz $\mathcal{M}^\Phi$, wenn gilt*

$$\mathcal{S} \triangle \mathcal{M}^S \models \Phi \triangle \mathcal{M}^\Phi$$

Ist $\mathcal{M}^\Phi$ die neutrale Modifikation, so liegt eine maskierende Fehlertoleranz *vor. Die auftretenden Fehler in $\mathcal{S}$ erhalten dann die Black-Box Eigenschaften des Systems.* □

Für eine sehr differenzierte Aussage über die Fehlertoleranz eines Systems kann unter Umständen der Nachweis mehrerer dieser Eigenschaften notwendig sein. Ein System kann bezüglich einer Abweichung $\mathcal{M}_1^\Phi$ die Fehler $\mathcal{M}_1^S$ tolerieren, bezüglich einer Abweichung $\mathcal{M}_2^\Phi$ die Fehler $\mathcal{M}_2^S$, und so weiter. In den meisten Fällen wird man sich aber auf ein einziges „maximales“ Fehlermodell konzentrieren und für diese Fehlerfälle die Auswirkungen ermitteln.

Wir werden in Definition 4.13 in Abschnitt 4.5.3 eine ähnliche Aussage wie in Definition 4.10 wiederfinden. Die Aussage der Fehlertoleranz stellt eine spezielle Interpretation der *korrespondierenden Modifikationen* dar. Mit den Beschreibungstechniken für Systeme und ihre Modifikationen zusammen mit den Beweistechniken können wir Fehlertoleranz ausdrücken und verifizieren, ohne neue Notationen oder Begriffe einführen zu müssen.

Auf einen Unterschied zu anderen Ansätzen zur Fehlermodellierung in der Literatur (insbesondere die Arbeiten von Joseph/Liu [48, 47] und Janowski [36, 35]) ist hier hinzuweisen, da er gerade in Bezug auf Fehlertoleranz eine Rolle spielt:

Janowski argumentiert, dass man sich bei Aussagen über Fehlertoleranz nicht auf das Auftreten von Fehlern verlassen darf: Fehler *können* auftreten, *müssen* es aber nicht. Er fordert den Nachweis, dass ein fehlertolerantes System die Toleranzspezifikation nicht nur bei Auftreten von Fehlern erfüllt, sondern auch dann, wenn Fehler nicht oder nur teilweise auftreten. In den Arbeiten der genannten Autoren werden Fehler in einem System nur durch zusätzliche Transitionen modelliert – das Entfernen von Transitionen wird nicht zugelassen. Zusätzliche Transitionen spiegeln tatsächlich einen temporären

Charakter von Fehlern besser wider: Die Fehlertransitionen können zwar jederzeit ausgeführt werden (wenn sie schaltbereit sind), sie müssen aber nicht schalten. Das System kann also immer noch ein fehlerfreies Verhalten zeigen und der Nachweis der Fehlertoleranz umfaßt wie gefordert auch immer alle Abläufe, in denen Fehler nicht auftreten.

Modifiziert man ein System nur durch Hinzufügen von neuen Transitionen, also durch eine Modifikation (E, F) mit $E = \varnothing$, so ist auch mit unserer Definition von Fehlertoleranz sichergestellt, dass das Auftreten von Fehlern in allen Abläufen nicht postuliert wird. Der Ansatz dieser Arbeit stimmt also in diesem Fall mit den Techniken der genannten Ansätze überein.

Es ist uns jedoch zusätzlich möglich, durch eine Modifikation das Entfernen von Transitionen durchzuführen. Dies kommt der Modellierung von permanenten Fehlern näher, da dem System manche Verhaltensmöglichkeiten von Anfang an genommen werden. Damit sind wir auch in der Lage, Fehlertoleranz auszudrücken unter der Annahme, dass Fehler sicher auftreten.

Unser Ansatz stellt damit einen allgemeineren Formalismus bereit, der bekannte Ansätze umfaßt. Wir ermöglichen damit einen allgemeineren Begriff von Fehlertoleranz, der Aussagen über veränderte Eigenschaften von veränderten Systemen macht. Durch die Wahl einer geeigneten Modifikation kann jede gewünschte Aussage formuliert werden.

Beispiel 4.8 Für die Systeme Merge^{Φ} und Merge^{S} aus den Beispielen 4.4 und 4.5 gilt

$$(\mathit{Merge}^{S} \triangle \mathcal{M}^{S}) \models (\mathit{Merge}^{\Phi} \triangle \mathcal{M}^{\Phi})$$

da sie das gleiche Verhalten beschreiben. Wir modifizieren $\mathit{Merge}^{S} \triangle \mathcal{M}^{S}$ durch das Hinzufügen von zwei weiteren Transitionen mittels $\mathcal{M}_2^S = (\varnothing, F)$ wobei

$F \stackrel{df}{=}$

Name	*Pre*	*Input*	*Output*	*Post*
τ_5	$c = 2$	$i_1?a$	$-$	$c' = c$
τ_6	$c = 1$	$i_2?a$	$-$	$c' = c$

Diese Transition lesen und verwerfen Daten nur von jeweils dem Eingabekanal, über den die (modifizierte) Black-Box Spezifikation keine Aussage macht. Das System $\mathit{Merge}^{S} \triangle \mathcal{M}^{S}$ ist also gegenüber dem Fehler $\mathcal{M}_2^S$ und der Eigenschaft $\mathit{Merge}^{\Phi} \triangle \mathcal{M}^{\Phi}$ *maskierend fehlertolerant*, da gilt

$$(\mathit{Merge}^{S} \triangle \mathcal{M}^{S}) \triangle \mathcal{M}_2^S \models (\mathit{Merge}^{\Phi} \triangle \mathcal{M}^{\Phi})$$

□

Im Beispiel wird deutlich, dass man auch an Modifikationen von bereits modifizierten Systemen interessiert ist. Im folgenden Abschnitt 4.5.1 behandeln wir dazu die Kombinationen von Modifikationen.

4.4.2 Fehlertoleranz als Dämpfung (quantitativ)

Wird ein System dem Einfluß von Fehlern ausgesetzt, so hat dies im allgemeinen Auswirkungen auf das Systemverhalten. Bei einem System, das Fehler nicht tolerieren kann, werden sich die Fehler auf schädliche Weise auf das Verhalten auswirken. Ist das System dagegen fehlertolerant, werden die Auswirkungen abgeschwächt und im Idealfall sogar vollkommen maskiert.

Die Fehler, denen ein System S ausgesetzt sein kann, können sowohl innerhalb des Systems (in einer Teilkomponente) als auch außerhalb (in der Systemumgebung) lokalisiert sein. Wir gehen im folgenden von einem zusammengesetzten System $S = S_1 \otimes S_2$ aus und betrachten einen Fehler in S_1. Dabei kann S_1 sowohl die Teilkomponente von S als auch die Umgebung von S_2 repräsentieren. Mit unserem Formalismus können wir diesen Fehler durch eine Modifikation $\mathcal{M}_1$ explizit angeben. Die durch diesen Fehler bewirkte Abweichung des Systemverhaltens kann durch eine Modifikation $\mathcal{M}$ von S formalisiert werden. In Abschnitt 4.5.2 werden wir Techniken kennen lernen, mit denen $\mathcal{M}$ aus $\mathcal{M}_1$ ermittelt werden kann. Es gilt dann

$$S \triangle \mathcal{M} = (S_1 \triangle \mathcal{M}_1) \otimes S_2$$

Der Grad der Fehlertoleranz läßt sich quantifizieren, wenn ein Maß für den Schweregrad von Modifikationen definiert werden kann. Sei d eine Funktion, die einer Modifikation eine entsprechende Maßzahl zuordnet. Das System S kann dann als fehlertolerant bezeichnet werden, wenn

$$d(\mathcal{M}) < d(\mathcal{M}_1)$$

Der korrekte Teil S_2 des Systems ist also in der Lage, die Auswirkungen des Fehlers innerhalb von S_1 einzudämmen.

Diese Charakterisierung von Fehlertoleranz basiert somit auf einer geeigneten Definition einer Bewertungsfunktion d von Modifikationen. Diese Funktion ist eng verbunden mit einer Metrik, die den Unterschied von Systemen quantifiziert, und über einen *Ähnlichkeitsbegriff* zwischen Systemverhalten zu definieren ist. Der Unterschied zwischen zwei Systemen wird der Bewertung d der Modifikation entsprechen, die S_1 in das System S_2 umwandelt. Es ist zu vermuten, dass der Schweregrad d einer Modifikation in Abhängigkeit vom System zu definieren ist, auf das sie angewendet wird. Sinnvolle Definitionen der genannten Begriffe verbleiben als offenes Problem.

Wir beschränken uns im folgenden auf eine qualitative Bewertung von Fehlerdämpfung. Mit dieser wird es möglich, die Fehlertoleranz von Systemen einer von fünf definierten Klassen zuzuordnen.

4.4.3 Fehlertoleranz als Dämpfung (qualitativ)

Mit Hilfe unseres Formalismus sind wir in der Lage, das Systemverhalten in die drei Klassen *Normal*, *Fehler* und *Chaos* zu unterteilen (vgl. Abbildung 4.1). Das Normalverhalten wird durch eine Beschreibung von S charakterisiert und das Fehlerverhalten durch $S \triangle \mathcal{M}$ beschrieben, während alle restlichen Verhalten als Chaos bezeichnet werden können.

Mit diesen drei Klassen können wir verschiedene Ausprägungen von Fehlerdämpfung definieren. Seien dazu wieder eine Modifikation $\mathcal{M}_1$ einer Teilkomponente bzw. der Umgebung und eine Modifikation $\mathcal{M}$ des Gesamtsystems gegeben. Diese implizieren eine Unterteilung des Verhaltens der fehlerhaften Teilkomponente (bzw. der Umgebung) und des Gesamtsystems in die drei genannten Klassen.

Wir können nun jeweils untersuchen, in welche Klasse das Systemverhalten für jede der drei Verhaltensklassen des fehlerhaften Teilsystems einzuordnen ist. Es gibt dabei die in der folgenden Tabelle dargestellten Möglichkeiten:

	(1)	(2)	(3a)	(3b)	(4)
Normal	Normal	Normal	Normal	Normal	Normal
Fehler	Normal	Normal	Fehler	Normal	Fehler
Chaos	Normal	Fehler	Fehler	Chaos	Chaos

In der Spalte ganz links sind die Verhaltensmöglichkeiten des fehlerverursachenden Teilsystems aufgeführt, in den anderen Spalten verschiedenen Möglichkeiten von Verhaltensklassen des Gesamtsystems. In Spalte (1) sind die Reaktionen eines *maximal fehlertoleranten* Systems dargestellt. Sogar wenn vollkommen unvorhergesehenes Verhalten in der Teilkomponente auftritt, bleibt das Gesamtverhalten normal. Spalte (2) beschreibt eine etwas schwächere Form der Fehlerdämpfung: Unvorhergesehene Fehler spiegeln sich nun in einem Fehlverhalten des Systems wider, während die vorhergesehenen und damit erwarteten Fehler aber noch keine Beeinträchtigung des Gesamtverhaltens nach sich ziehen. Nach den Zwischenformen (3a) und (3b) charakterisiert Spalte (4) die schwächste Form der Dämpfung von Fehlereinflüssen, die – genaugenommen – keine wirkliche Dämpfung mehr darstellt: Fehler führen wieder zu sichtbaren Fehlern des Gesamtsystems, chaotisches Verhalten der Teilkomponente bewirkt chaotisches Gesamtverhalten.

Die Tabelle ist vollständig in dem Sinne, dass andere Kombinationen nicht mehr sinnvoll als Fehlertoleranz verstanden werden können. So darf das Gesamtverhalten keine Fehler aufweisen, wenn die betrachtete Teilkomponente korrekt ist; die erste Zeile darf also nur die Einträge *Normal* enthalten. Das Gesamtverhalten darf nicht ins Chaos verfallen, wenn die Komponente vorhergesehene Fehler zeigt, also kann *Chaos* nicht in der zweiten Zeile erscheinen. Ein System, das auf vorhergesehene Fehler mit sichtbaren Fehlern reagiert, auf unvorhergesehene Fehler aber mit Normalverhalten reagiert, stellt einen unplausiblen Sonderfall dar, den wir nicht in unsere Untersuchungen aufnehmen.

Wir können die verschiedenen Arten von Fehlertoleranz durch eine Ordnung $>$ vergleichen, die wir durch

$$(1) \quad > \quad (2) \quad > \quad \begin{matrix} (3a) \\ (3b) \end{matrix} \quad > \quad (4)$$

definieren. Eine „größere“ Fehlertoleranz eines Systems besagt informell, dass *größere* Fehler eine *geringere* Auswirkungen haben.

Die angegebene Klassifikation eignet sich also, bei gegebenen Modifikationen Systeme bezüglich ihrer Fähigkeit zur Fehlerdämpfung zu bewerten.

Die qualitative Bewertung der Fehlerdämpfung eines Systems ist relativ zur Definition der Modifikationen. So ist durch die Wahl von $\mathcal{M}$ beliebig definierbar, ob ein fehlerinduziertes Gesamtverhalten als vorhersehbarer Fehler oder als chaotisches Verhalten betrachtet wird. Wählt man beispielsweise als Modifikation die Ergänzung zu beliebigem Verhalten, so kann der Fall *Chaos* nicht mehr auftreten, und jeder Fehler wird als

„vorhergesehen“ qualifiziert; es liegt also mindestens eine Dämpfung entsprechend (3a) vor. Wählt man eine neutrale Modifikation zur Definition der Fehler des Gesamtsystems, wird jegliche Abweichung vom Normalverhalten als *Chaos* bewertet; die Klasse *Fehler* tritt nicht mehr auf. Es kommen dann zur Bewertung der Dämpfung nur noch die beiden Klassen (1) und (3b) in Frage, und eine Bewertung von Systemen bezüglich ihrer Fehlerdämpfung kann nur noch sehr grob ausfallen. Die Wahl der Modifikationen mit der verbundenen Einteilung des Verhaltens in die genannten drei Klassen hat also sorgfältig zu erfolgen, um eine sinnvolle Bewertung zu ermöglichen.

Die Charakterisierung von Fehlertoleranz entsprechend Definition 4.10 aus Abschnitt 4.4.1 als korrespondierende Modifikationen entspricht den Varianten (3a) und (4) der Fehlerdämpfung: Die Abwesenheit von Fehlern bewirkt Normalverhalten, auftretende Fehler bewirken Fehlverhalten des Gesamtsystems und über die Auswirkungen unvorhergesehener Fehler wird keine Aussage gemacht.

Typischerweise hat ein System nicht die gewünschten Eigenschaften von Fehlertoleranz, da es möglicherweise noch ohne Berücksichtigung potentieller Fehler entwickelt wurde. Nach der Identifikation der zu erwartenden Fehler und ihrer Formalisierung durch Modifikationen ist das System im weiteren Verlauf des Entwicklungsprozesses um entsprechende Mechanismen der Fehlertoleranz zu erweitern. Entsprechende Entwicklungsschritte werden in Kapitel 5 vorgestellt.

4.5 Eigenschaften von Modifikationen

Wir werden in diesem Abschnitt weitere Konzepte und Eigenschaften definieren, die auf Basis des vorgestellen Modifikationsbegriffes möglich geworden sind. Dazu zeigen wir, wie mehrere Modifikationen zu einer Modifikation kombiniert werden können und untersuchen ihre Wirkung über verschiedene Beschreibungsebenen hinaus. Wir diskutieren die Zusammenhänge von Modifikationen bezüglich verschiedener Beschreibungstechniken unter dem Stichwort der korrespondierenden Modifikationen.

4.5.1 Kombination von Modifikationen

Im Laufe eines Entwicklungsprozesses ist es möglich, dass ein System mehreren Modifikationen ausgesetzt wird. Es ist auf dem Weg zu einem realistischen System in einer ebenso realistischen, also unter Umständen fehlerbehafteten Umgebung nicht immer von Anfang an in vollem Umfang bekannt, wie ein System angepasst werden muss. Es sind stattdessen mehrere Iterationen möglich: Eine Komponente wird verändert, was die Modifikation einer anderen Komponente nach sich zieht, was wiederum eine Veränderung der ersten Komponente möglich macht, und so weiter.

Wir wollen hier nun formal ausdrücken, wie mehrere Modifikationen zu einer Modifikation zusammengefaßt werden können – sowohl im Formalismus der Black-Box Spezifikation als auch im Rahmen von Transitionssystemen.

Wird die Black-Box Eigenschaft einer Komponente zweimal modifiziert, also je zweimal verstärkt und abgeschwächt, so entspricht dies einer Modifikation durch die Konjunktion der Verstärkungen, aber einer Disjunktion der Abschwächungen:

Definition 4.11 *Kombination von Black-Box Modifikationen*

Die Kombination zweier Black-Box Modifikationen $\mathcal{M}_1$ und $\mathcal{M}_2$ wird mit Hilfe des Operators + definiert durch:

$$(\Phi_E^1, \Phi_F^1) + (\Phi_E^2, \Phi_F^2) \stackrel{df}{=} (\Phi_E^1 \wedge \Phi_E^2, \Phi_F^1 \vee \Phi_F^2)$$

□

Sind die beiden Modifikationen *unabhängig* voneinander, können sie in beliebiger Reihenfolge, und sogar gemeinsam als kombinierte Modifikation auf ein System angewendet werden, wobei sich immer das gleiche Resultat ergibt. Dies formulieren wir folgendermaßen:

Proposition 4.5 *Unabhängigkeit von Black-Box Modifikationen*

Der Operator + für Black-Box-Spezifikationen ist assoziativ und kommutativ. Dies ergibt sich unmittelbar aus der Assoziativität und Kommutativität von $\wedge$ und $\vee$.

Unter der Voraussetzung der Unabhängigkeit

$$\Phi_F^1 \Rightarrow \Phi_E^2 \quad \wedge \quad \Phi_F^2 \Rightarrow \Phi_E^1$$

gilt für $\mathcal{M}_1 = (\Phi_E^1, \Phi_F^1)$ und $\mathcal{M}_2 = (\Phi_E^2, \Phi_F^2)$, dass die Reihenfolge der Anwendung der Modifikationen $\mathcal{M}_1$ und $\mathcal{M}_2$ keine Auswirkung auf das Resultat hat, und identisch ist mit dem System, das durch die Kombination beider modifiziert wurde:

$$(\Phi \triangle \mathcal{M}_1) \triangle \mathcal{M}_2 \quad = \quad \Phi \triangle (\mathcal{M}_1 + \mathcal{M}_2) \quad = \quad (\Phi \triangle \mathcal{M}_2) \triangle \mathcal{M}_1$$

Dies wird einfach gezeigt unter Ausnutzung der Annahmen, die die Äquivalenzen $\Phi_F^1 \wedge \Phi_E^2 \Leftrightarrow \Phi_F^1$ und $\Phi_F^2 \wedge \Phi_E^1 \Leftrightarrow \Phi_F^2$ implizieren:

$$\begin{aligned}
&(\Phi \triangle \mathcal{M}_1) \triangle \mathcal{M}_2 \\
&\qquad = (((\Phi \wedge \Phi_E^1) \vee \Phi_F^1) \wedge \Phi_E^2) \vee \Phi_F^2 \\
&\qquad = (\Phi \wedge \Phi_E^1 \wedge \Phi_E^2) \vee (\Phi_F^1 \wedge \Phi_E^2) \vee \Phi_F^2 \\
&\qquad = (\Phi \wedge (\Phi_E^1 \wedge \Phi_E^2)) \vee (\Phi_F^1 \vee \Phi_F^2) \\
&\qquad = \Phi \triangle ((\Phi_E^1, \Phi_F^1) + (\Phi_E^2, \Phi_F^2)) \\
&= \Phi \triangle (\mathcal{M}_1 + \mathcal{M}_2) = \Phi \triangle (\mathcal{M}_2 + \mathcal{M}_1) \\
&\qquad = \Phi \triangle ((\Phi_E^2, \Phi_F^2) + (\Phi_E^1, \Phi_F^1)) \\
&\qquad = (\Phi \wedge (\Phi_E^2 \wedge \Phi_E^1)) \vee (\Phi_F^2 \vee \Phi_F^1) \\
&\qquad = (\Phi \wedge \Phi_E^2 \wedge \Phi_E^1) \vee (\Phi_F^2 \wedge \Phi_E^1) \vee \Phi_F^1 \\
&\qquad = (((\Phi \wedge \Phi_E^2) \vee \Phi_F^2) \wedge \Phi_E^1) \vee \Phi_F^1 \\
&= (\Phi \triangle \mathcal{M}_2) \triangle \mathcal{M}_1
\end{aligned}$$

□

Die Forderung der Unabhängigkeit $\Phi_F^1 \Rightarrow \Phi_E^2$ (und umgekehrt) besagt durch Ausschluß des Falles $\Phi_F^1 \wedge \neg \Phi_E^2$, dass durch Abschwächung der Spezifikation durch eine der beiden Modifikationen ein neu gestattetes Verhalten nicht im Widerspruch zur Eigenschaft stehen darf, die durch die jeweils andere Modifikation eingefordert wird. Eine

neutrale Modifikation $(\mathsf{true}, \mathsf{false})$ ist immer unabhängig von einer beliebigen anderen Modifikation, da $\mathsf{false} \Rightarrow \Phi_E$ und $\Phi_F \Rightarrow \mathsf{true}$ immer gelten.

Modifikationen von *Transitionssystemen* lassen sich kombinieren, indem wir die Transitionen, die mittels zweier Kombinationen hinzugefügt (bzw. entfernt) werden, auf einmal hinzufügen (bzw. entfernen). Wir verwenden wieder den Operator $+$, der damit überladen wird:

Definition 4.12 *Kombination von Modifikationen von Transitionssystemen*

Die Kombination zweier Modifikationen $\mathcal{M}_1$ und $\mathcal{M}_2$ eines Transitionssystems wird definiert durch

$$(E_1, F_1) + (E_2, F_2) \stackrel{df}{=} (E_1 \cup E_2, F_1 \cup F_2)$$

□

Eine Unabhängigkeit von Modifikationen liegt dann vor, wenn vermieden wird, dass eine Modifikation Transitionen ergänzt, die von der anderen entfernt werden. Es gilt dann die folgende Eigenschaft:

Proposition 4.6 *Unabhängigkeit von Modifikationen von Transitionssystemen*

Der Operator $+$ für Modifikationen von Transitionssystemen ist assoziativ und kommutativ, wie sich unmittelbar aus der Assoziativität und Kommutativität der Mengenvereinigung $\cup$ ergibt.

Unter der Voraussetzung

$$E_1 \cap F_2 = \varnothing \quad \wedge \quad E_2 \cap F_1 = \varnothing$$

für Modifikationen $\mathcal{M}_1 = (E_1, F_1)$ und $\mathcal{M}_2 = (E_2, F_2)$ gilt

$$(\mathcal{S} \triangle \mathcal{M}_1) \triangle \mathcal{M}_2 \quad = \quad \mathcal{S} \triangle (\mathcal{M}_1 + \mathcal{M}_2) \quad = \quad (\mathcal{S} \triangle \mathcal{M}_2) \triangle \mathcal{M}_1$$

Die Voraussetzung stellt sicher, dass $F_1 \cap E_2 = \varnothing = F_2 \cap E_1$, so dass (mit T als Transitionsmenge von S) die Behauptung bewiesen wird durch

$$\begin{aligned}
&(T \triangle \mathcal{M}_1) \triangle \mathcal{M}_2 \\
&\qquad = ((T \setminus E_1) \cup F_1) \setminus E_2) \cup F_2 \\
&\qquad = ((T \setminus E_1) \setminus E_2) \cup (F_1 \setminus E_2)) \cup F_2 \\
&\qquad = (T \setminus (E_1 \cup E_2)) \cup (F_1 \cup F_2) \\
&= T \triangle (\mathcal{M}_1 + \mathcal{M}_2) = T \triangle (\mathcal{M}_2 + \mathcal{M}_1) \\
&\qquad = (T \setminus (E_2 \cup E_1)) \cup (F_2 \cup F_1) \\
&\qquad = ((T \setminus E_2) \setminus E_1) \cup (F_2 \setminus E_1)) \cup F_1 \\
&\qquad = ((T \setminus E_2) \cup F_2) \setminus E_1) \cup F_1 \\
&= (T \triangle \mathcal{M}_2) \triangle \mathcal{M}_1
\end{aligned}$$

□

Wir illustrieren die Notwendigkeit der Unabhängigkeitsforderung an folgendem Beispiel.

Beispiel 4.9 Kombination von Modifikationen von Transitionssystemen

Wir betrachten die Komponente *Merge* aus Beispiel 4.5, die durch $\mathcal{M}$ so verändert wurde, dass sie nur noch von einem Kanal liest.

Die Modifikation $\mathcal{M}'$ soll den Verlust einzelner Nachrichten auf den Eingabekanälen modellieren. Dies wird spezifiziert durch $(\varnothing, F')$ mit

$F' \stackrel{df}{=}$

Name	*Pre*	*Input*	*Output*	*Post*
τ_5		$i_1?d$	–	–
τ_6		$i_2?d$	–	–

Von der Kombination $\mathcal{M} + \mathcal{M}'$ erwarten wir, dass sie zu einer Komponente führt, die nur noch von einem Kanal liest, aber auf diesem auch Nachrichten verlieren kann.

Es liegt aber keine Unabhängigkeit der Modifikationen vor, da durch $\mathcal{M}$ mittels E_1 alle Transitionen entfernt werden, die c verändern; die neuen Transitionen in F' enthalten aber die Variable c nicht, und können sie daher auf beliebige Werte setzen. Die Eigenschaften aus Proposition 4.6 gelten hier in der Tat nicht: Während $(Merge \triangle \mathcal{M}') \triangle \mathcal{M}$ das erwartete Verhalten beschreibt, kann sich $(Merge \triangle \mathcal{M}) \triangle \mathcal{M}'$ nach dem Verlust eines einzelnen Datums für einen neuen Kanal c entscheiden. Wählen wir $c' = c$ als Nachbedingung von τ_5 und τ_6, können wir einen Konflikt zwischen $\mathcal{M}$ und $\mathcal{M}'$ vermeiden. □

Die *Modifikationskomponente*, die sich durch die Kombination zweier Modifikationskomponenten ergibt, ist naheliegend: Sie wird durch die normale Komponentenkomposition $\otimes$ unseres Systemmodells dargestellt, also durch $\mathcal{M} = \mathcal{M}_1 \otimes \mathcal{M}_2$. Das Ergebnis ist oft von einem allgemeineren Typ als der Typ der einzelnen Modifikationskomponenten, und führt meist zu dem in Abbildung 4.3(d) dargestellten Fall. Modifiziert beispielsweise $\mathcal{M}_1$ die Eingabekanäle, aber $\mathcal{M}_2$ die Ausgabekanäle, so ist $\mathcal{M}$ nur durch die Variante darstellbar, die beide Kanaltypen modifiziert.

4.5.2 Fortpflanzung von Modifikationen

Wird ein System modifiziert, so hat dies selbstverständlich auch Auswirkungen auf die Umgebung des Systems: Zeigt das System neues Verhalten, so wird die Umgebung mit neuem Verhalten darauf reagieren; zeigt das System im Gegenteil ein bestimmtes Verhalten nicht mehr, wird auch die Umgebung ein entsprechend anderes, meist reduziertes Verhalten aufweisen.

Da wir Änderungen im Verhalten von Systemen durch Modifikationen ausdrücken können, sind wir in der Lage, die Auswirkungen der Modifikation eines Systemteils erneut als Modifikation des Gesamtsystems auszudrücken. Wir gehen im folgenden entsprechend Abbildung 4.5 von einem zusammengesetzten System S aus, das aus zwei Komponenten S_1 und S_2 besteht, d.h.

$$S \rightsquigarrow S_1 \otimes S_2$$

Aufgrund der Eigenschaften unseres Systemmodells sind wir durch die Abstützung der folgenden Definition auf nur zwei Komponenten keineswegs in der Anwendbarkeit eingeschränkt, da sich beliebig viele Komponenten immer durch *eine* Komponente ausdrücken lassen, die aus ihnen zusammengesetzt wird. So kann S_1 tatsächlich nur eine

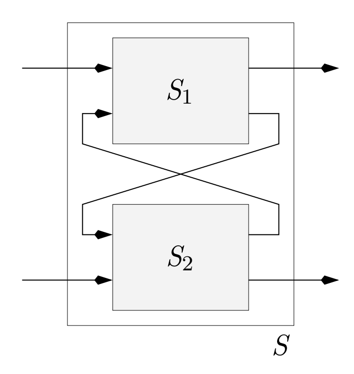


Abbildung 4.5: Zusammengesetztes System

Komponente sein, während S_2 die Komposition der restlichen Komponenten beinhaltet. Wir wollen nun ermitteln, welche Modifikation des Gesamtverhaltens sich aus einer Modifikation von S_1 ergibt.

Wir können für die Darstellungsform der Black-Box Spezifikationen als auch der Transitionssysteme Charakterisierungen der Auswirkungen definieren. Dazu gehen wir jeweils von einem zusammengesetzten System aus und definieren aus den Modifikationen der Einzelkomponenten die äquivalente Modifikation des Gesamtsystems.

Proposition 4.7 *Fortpflanzung von Modifikationen bei Black-Box Spezifikationen*

Sei ein System S durch die Black-Box Spezifikation Φ gegeben, das durch ein komponiertes System aus zwei Komponenten S_1 und S_2 verfeinert wird. Das Verhalten der beiden Komponenten wird jeweils durch eine Black-Box Spezifikation Φ_1 und Φ_2 beschrieben, d.h.

$$\Phi \rightsquigarrow \Phi_1 \otimes \Phi_2 \quad \textit{oder auch} \quad \Phi_1 \wedge \Phi_2 \Rightarrow \Phi$$

Wird das Verhalten der Komponenten durch Modifikationen $\mathcal{M}_1 = (\Phi_E^1, \Phi_F^1)$ und $\mathcal{M}_2 = (\Phi_E^2, \Phi_F^2)$ verändert, so entspricht dies der Modifikation $\mathcal{M} = (\Phi_E, \Phi_F)$ des Gesamtsystems mit

$$\begin{aligned} \Phi_E &= \ \Phi_E^1 \wedge \Phi_E^2, \\ \Phi_F &= (\Phi^1 \wedge \Phi_F^2) \vee (\Phi^2 \wedge \Phi_F^1) \vee (\Phi_F^1 \wedge \Phi_F^2) \end{aligned}$$

Es gilt dann also

$$\Phi \triangle \mathcal{M} \rightsquigarrow (\Phi_1 \triangle \mathcal{M}_1) \otimes (\Phi_2 \triangle \mathcal{M}_2)$$

Dies kann leicht bewiesen werden, wie folgende Deduktion zeigt:

$$
\begin{aligned}
&(\Phi_1 \triangle \mathcal{M}_1) \wedge (\Phi_2 \triangle \mathcal{M}_2) \\
&\quad = ((\Phi_1 \wedge \Phi_E^1) \vee \Phi_F^1) \wedge ((\Phi_2 \wedge \Phi_E^2) \vee \Phi_F^2) \\
&\quad = (\Phi_1 \wedge \Phi_E^1 \wedge \Phi_2 \wedge \Phi_E^2) \vee (\Phi_1 \wedge \Phi_E^1 \wedge \Phi_F^2) \vee \\
&\qquad\qquad (\Phi_F^1 \wedge \Phi_2 \wedge \Phi_E^2) \vee (\Phi_F^1 \wedge \Phi_F^2) \\
&\quad \Rightarrow (\Phi_1 \wedge \Phi_E^1 \wedge \Phi_2 \wedge \Phi_E^2) \vee (\Phi_1 \wedge \Phi_F^2) \vee \\
&\qquad\qquad (\Phi_F^1 \wedge \Phi_2) \vee (\Phi_F^1 \wedge \Phi_F^2) \\
&\quad \Rightarrow \Phi \triangle (\Phi_E, \Phi_F)
\end{aligned}
$$

□

Eine resultierende Modifikation des Gesamtsystems läßt sich auch informell interpretieren: Die Eigenschaft von S wird verstärkt um die Konjunktion der zusätzlichen Eigenschaften, die von den modifizierten Komponenten erfüllt werden müssen, und die durch die Komposition nicht ungültig gemacht werden. Durch neue als Φ_F formulierte Verhaltensmöglichkeiten hat S zusätzliche Freiheiten in seinem Verhalten: Es darf einerseits neue Verhaltensmöglichkeiten einer Teilkomponenten kombinieren mit dem unveränderten Verhalten der jeweils anderen Teilkomponente, andererseits darf es auch die neuen Verhaltensmöglichkeiten beider Teilkomponenten kombiniert ausnützen.

Gilt die Annahme der Unabhängigkeit der Modifikationen, wie wir sie bereits in Abschnitt 4.5.1 kennengelernt haben, also

$$\Phi_F^1 \Rightarrow \Phi_E^2 \quad \text{und} \quad \Phi_F^2 \Rightarrow \Phi_E^1$$

und zeigt $S_1 \otimes S_2$ *genau* das Verhalten von S, gilt also auch $\Phi_1 \otimes \Phi_2 \rightsquigarrow \Phi$, so können im Beweis die Implikationen ersetzt werden zu Gleichheiten und die Aussage der Proposition 4.7 verstärkt werden zu

$$\Phi \triangle \mathcal{M} = (\Phi_1 \triangle \mathcal{M}_1) \otimes (\Phi_2 \triangle \mathcal{M}_2)$$

Auch für Transitionssysteme können wir eine Modifikation eines Systems $\mathcal{S}$ finden, die zu Modifikation von Teilkomponenten paßt. Da wir aber keinen Verfeinerungsbegriff zwischen Transitionssystemen definiert haben (wie in Abschnitt 3.9 begründet), definieren wir die Aussage über die Gleichheit von modifizierten, zusammengesetzten Systemen.

Proposition 4.8 *Fortpflanzung von Modifikationen bei Transitionssystemen*

Seien zwei kompatible Transitionssysteme $\mathcal{S}_1$ und $\mathcal{S}_2$ gegeben, und sei ihre Komposition $\mathcal{S}_1 \otimes \mathcal{S}_2$ definiert wie in Abschnitt 3.7, insbesondere sei die Transitionsmenge T des komponierten System also gegeben durch

$$T = (T_1 \bowtie T_2^\epsilon) \cup (T_1^\epsilon \bowtie T_2)$$

Seien weiterhin $\mathcal{M}_1 = (E_1, F_1)$ und $\mathcal{M}_2 = (E_2, F_2)$ Modifikationen der beiden Teilsysteme. Die Modifikation $\mathcal{M} = (E, F)$, definiert durch

$$
\begin{aligned}
E &= (E_1 \bowtie T_2^\epsilon) \cup (E_2 \bowtie T_1^\epsilon) \\
F &= (F_1 \bowtie T_2^\epsilon) \cup (F_2 \bowtie T_1^\epsilon)
\end{aligned}
$$

beschreibt damit die entsprechende Modifikation des zusammengesetzten Systems, so dass also gilt

$$(\mathcal{S}_1 \otimes \mathcal{S}_2)\triangle(E, F) = (\mathcal{S}_1\triangle(E_1, F_1)) \otimes (\mathcal{S}_2\triangle(E_2, F_2))$$

Zum Nachweis dieser Aussage expandiert man die entsprechenden Definitionen zu

$$(((T_1 \bowtie T_2^\epsilon) \cup (T_2 \bowtie T_1^\epsilon)) \setminus ((E_1 \bowtie T_2^\epsilon) \cup (E_2 \bowtie T_1^\epsilon))) \cup (F_1 \bowtie T_2^\epsilon) \cup (F_2 \bowtie T_1^\epsilon)$$
$$=$$
$$(((T_1 \setminus E_1) \cup F_1) \bowtie T_2^\epsilon) \cup (((T_2 \setminus E_2) \cup F_2) \bowtie T_1^\epsilon)$$

und zeigt dies unter Verwendung der folgenden Gleichheiten für die Operatoren $op \in \{\setminus, \cup\}$.

$$(X \bowtie Z)\ op\ (Y \bowtie Z) = (X\ \ op\ \ Y) \bowtie Z$$

□

Der vorgestellte Formalismus kann verwendet werden, um die Auswirkungen zu berechnen, die die Fehler in den Komponenten eines Systems haben. Wir illustrieren dies mit folgendem Beispiel.

Beispiel 4.10 Fehlerfortpflanzung

In Abschnitt 3.7 wurden die Komponenten *Merge* und *Split* zu einem System *Multiplex* komponiert, von dem wir bereits die Eigenschaft

$$\Phi = \begin{array}{l} D_1 Ⓢ o = i_1 \wedge D_2 Ⓢ o = i_2 \\ D_1 Ⓢ o = o_1 \wedge D_2 Ⓢ o = o_2 \end{array}$$

nachgewiesen haben. In Beispiel 4.4 haben wir eine Modifikation von *Merge* definiert, die nur noch von einem Kanal liest und den anderen ignoriert. Wir ermitteln nun, welche Auswirkung die Änderung von *Merge* auf das zusammengesetzte System *Multiplex* hat, bei unmodifizierter Komponente *Split*. Die Modifikationen der Komponenten lauten also

$$\begin{array}{ll} \mathcal{M}_{Merge} = & (\mathsf{false}, o = i_1 \vee o = i_2) \\ \mathcal{M}_{Split} = & (\mathsf{true}, \mathsf{false}) \end{array}$$

Die (noch weiter zu vereinfachende) zusammengesetzte Modifikation $\mathcal{M} = \mathcal{M}_{Merge} + \mathcal{M}_{Split}$ ergibt sich nach Definition 4.11 zu

$$\begin{array}{ll} \mathcal{M} = (& \mathsf{false} \wedge \mathsf{true}, \\ & \mathsf{false} \vee ((D_1 Ⓢ o = o_1 \wedge D_2 Ⓢ o = o_2) \wedge (o = i_1 \vee o = i_2)) \vee \mathsf{false} \ \ ) \end{array}$$

Wir müssen nun nicht die Modifikationen der Einzelkomponenten einzeln anwenden, sondern können diese kombinierte Modifikation unmittelbar auf die Eigenschaft Φ des Gesamtsystems anwenden, und erhalten

$$\Phi\triangle\mathcal{M} = \ (\Phi \wedge \mathsf{false}) \vee ((D_1 Ⓢ o = o_1 \wedge D_2 Ⓢ o = o_2) \wedge (o = i_1 \vee o = i_2))$$

Aus dem ersten Fall $o = i_1$ der darin enthaltenen Disjunktion können wir $D_1 Ⓢ o = o$ ableiten, und daraus sowohl $o_1 = D_1 Ⓢ o = o$ als auch $\langle\rangle = D_2 Ⓢ o = o_2$ unter Ausnutzen der Disjunktheit von D_1 und D_2. Mit einem ähnlichen Argument für den zweiten Fall können wir schließlich für *Multiplex* mit dem modifizierten *Merge* die (zu erwartende) Eigenschaft herleiten, dass nur auf genau einem der beiden Ausgabekanäle Nachrichten ausgegeben werden:

$$(o_1 = i_1 \wedge o_2 = \langle\rangle) \quad \vee \quad (o_2 = i_2 \wedge o_1 = \langle\rangle)$$

□

4.5.3 Korrespondierende Modifikationen

Im vorherigen Abschnitt über die Fortpflanzung von Modifikationen wurde behandelt, wie sich Modifikationen von Systemkomponenten auf das Gesamtsystem auswirken. Diese Auswirkung ließ sich wiederum als Modifikation des Gesamtsystems ausdrücken. Es wurden sowohl die Komponenten als auch das System, das sie bilden, im gleichen Formalismus beschrieben, also jeweils durch Black-Box Spezifikationen oder durch Transitionssysteme.

In diesem Abschnitt behandeln wir den Zusammenhang von Modifikationen eines Systems, das zweimal mit Hilfe verschiedener Techniken beschrieben wird, also sowohl durch Black-Box Eigenschaften als auch durch Transitionssysteme. Wir nennen Modifikationen der beiden Sichten *korrespondierend*, wenn ihre Wirkungen sich entsprechen. Man kann den Bedarf nach dem Begriff der korrespondierenden Modifikationen auf verschiedene Arten motivieren:

- Im Laufe eines Entwicklungsprozesses kann sich eine Änderung der Anforderungen an ein System ergeben, beispielsweise durch eine veränderte Umgebung, durch das Auftauchen neuer Kundenwünsche oder durch eine Korrektur, die eine fehlerhafte Spezifikation nötig macht. Hat man aber bereits eine (abstrakte) Implementierung des Systems in Form eines Transitionssystems, so muß dieses an die neuen Forderungen angepasst werden.
- Auch umgekehrt können sich Modifikationen ergeben: Werden beispielsweise Fehler in einer Implementierung beobachtet, so lassen sich diese in vielen konkreten Fällen leicht durch zusätzliche Transitionen nachmodellieren. Spontane Änderungen von Daten oder der Verlust und die Verfälschung von Daten bei der Übertragung sind auf diese Weise gut darstellbar. Man ist in einem solchen Fall daran interessiert, wie sich die (Black-Box) Eigenschaften des so modifizierten Systems verändern, also ob zum Beispiel sicherheitskritische Eigenschaften verletzt werden.

Die Idee der korrespondierenden Modifikationen wird in Abbildung 4.6 dargestellt. Ein Transitionssystem $\mathcal{S}$ habe eine Eigenschaft Φ. Beide Beschreibungen des Systems werden passend modifiziert, so dass das modifizierte System die modifizierte Eigenschaft hat. Wir definieren dies formal:

Definition 4.13 *Korrespondierende Modifikationen*

Ein Transitionssystem $\mathcal{S}$ habe eine Eigenschaft Φ, also $\mathcal{S} \models \Phi$. Ferner sei $\mathcal{M}^{\mathcal{S}}$ eine Modifikation von $\mathcal{S}$ und $\mathcal{M}^{\Phi}$ eine Modifikation von Φ.

Die beiden Modifikationen $\mathcal{M}^{\mathcal{S}}$ und $\mathcal{M}^{\Phi}$ heißen korrespondierende Modifikationen zu $\mathcal{S}$ und Φ*, wenn gilt*

$$\mathcal{S} \triangle \mathcal{M}^{S} \quad \models \quad \Phi \triangle \mathcal{M}^{\Phi}$$

□

Diese Definition von korrespondierenden Modifikationen ist relativ zu konkreten $\mathcal{S}$ und Φ. Dies ist begründet durch die Beobachtung, dass korrespondierende Modifikationen

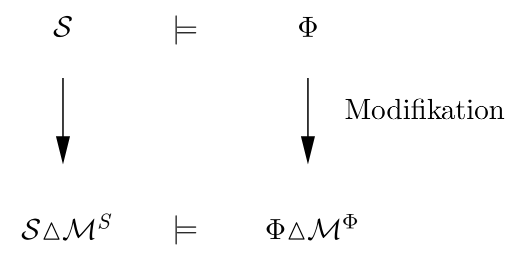


Abbildung 4.6: Korrespondierende Modifikationen

im allgemeinen nicht auf beliebige Systeme anwendbar sind. Ein offensichtlicher Grund ist, dass die Modifikationen von Transitionssystemen konkreten Bezug auf die Namen von Variablen nehmen. Zwei Transitionssysteme, die sich nur in der Wahl der Namen für interne Variablen unterscheiden, haben natürlich die gleichen Eigenschaften, würden aber verschiedene Formulierungen von Modifikationen benötigen. Dieses Problem ließe sich noch beheben durch entsprechende Formalismen, die gegenüber Variablennamen indifferent sind (die aber die praktische Handhabung erschweren würden). Dennoch kann beispielsweise das Hinzufügen der Transition

Name	*Pre*	*Input*	*Output*	*Post*
τ	–	$i_1?d$	–	–

in Systemen mit gleicher Schnittstelle, aber völlig anderer Funktionalität vollkommen unterschiedliche Auswirkungen haben: Stellt dies in der Komponente *Merge* den unbemerkt bleibenden Verlust eines Datums dar, könnte dies in einer anderen Komponente den Verlust einer Quittierung darstellen, die zum Beispiel das Rücksetzen der Komponente bewirkt, um nur ein willkürliches Beispiel zu nennen. Wir beziehen korrespondierende Modifikationen daher im allgemeinen sinnvollerweise auf konkrete Systeme.

Es ist eine komplexe Aufgabe, in konkreten Anwendungsfällen korrespondierende Modifikationen zu finden. Hat man also bereits einen Beweis für $\mathcal{S} \models \Phi$ gefunden, und modifiziert man eine der beiden Systembeschreibungen, so ergeben sich zwei Fragestellungen:

- *Wie lautet die zu einer vorgegeben Modifikation passende korrespondierende Modifikation?*

  Ideal wäre ein konstruktives Verfahren, das zu gegebenen Systembeschreibungen und einer Modifikation die entsprechende korrespondierende Modifikation angibt. Leider ist dies im allgemeinen nicht möglich, da man sonst auch in der Lage wäre, aus einem beliebigen Transitionssystem $\mathcal{S}$ automatisch auf seine Eigenschaften Φ zu schließen: Man würde ein beliebiges System mit einfachen Eigenschaften Φ' wählen und es zum gewünschten System $\mathcal{S}$ modifizieren. Ein konstruktives Verfahren würde die Modifikation $\mathcal{M}^\Phi$ liefern, und $\Phi'\triangle\mathcal{M}^\Phi$ würde die gesuchten Eigenschaften repräsentieren.

  Es ist zu vermuten, dass eine korrespondierende Modifikation nur von einem Entwickler des Systems gefunden werden kann, der ein Verständnis für die Funktionsweise des Systems hat und die Auswirkungen von Modifikationen abschätzen

kann. Eine Black-Box Spezifikation stellt auch immer eine Abstraktion dar, und umfaßt in den meisten Fällen nicht wirklich alle Eigenschaften, die ein Transitionssystem aufweist, sondern beschränkt sich auf Eigenschaften, die im Sinne der Anwendung interessant sind. Auf die gleiche Weise, wie zueinander passende Spezifikationen und Transitionssysteme durch Verständnis, Kreativität und Intuition gefunden werden müssen, müssen auch korrespondierende Modifikationen gefunden werden.

Würden sich korrespondierende Modifikationen konstruktiv finden lassen, wäre die Korrektheit der Korrespondenz automatisch gegeben. Da wir sie aber selbst finden müssen, muss noch ihre Korrektheit gezeigt werden. Als Frage resultiert daraus:

- *Wie beweist man die Korrektheit der modifizierten Eigenschaft?*

  Selbstverständlich läßt sich die Aussage $\mathcal{S} \triangle \mathcal{M}^S \models \Phi \triangle \mathcal{M}^\Phi$ mit den gleichen Methoden beweisen, wie wir sie in Abschnitt 3.8 dargestellt haben. Allerdings kann der Aufwand relativ hoch sein, den Beweis vollkommen neu zu führen. Wünschenswerter ist es, den bereits existierenden Beweis für $\mathcal{S} \models \Phi$ ausnützen zu können, um den darin bereits investierten Aufwand nicht zu verlieren. Wir wollen also nach Möglichkeit bei Modifikationen von Systemen auch ihre Beweise entsprechend modifizieren. Da wir Beweise durch Diagramme repräsentieren, können wir eine Beweismodifikation durch eine Anpassung der Beweisdiagramme darstellen. Wir werden dies im Abschnitt 5.7 diskutieren.

Korrespondierende Modifikationen können zur Formulierung von Fehlertoleranzaussagen verwendet werden. Wie in Kapitel 2.2 beschrieben, sind für eine präzise Aussage die Fehler und die tolerierbaren Auswirkungen explizit zu beschreiben. Mit $\mathcal{M}^S$ als Fehlermodell einer Implementierung und $\mathcal{M}^\Phi$ zum Formulieren veränderter Eigenschaften stehen geeignete Ausdrucksmöglichkeiten bereit.

Beispiel 4.11 Korrespondierende Modifikationen

Wir haben in Beispiel 3.3 die Komponente Merge^S als Transitionssystem definiert, und in Beispiel 3.5 seine Black-Box Eigenschaften Merge^Φ. Die Modifikation, nur noch von einem der beiden Kanäle zu lesen, wurde in beiden Formalismen in Beispiel 4.4 ($\mathcal{M}^\Phi$) und Beispiel 4.5 ($\mathcal{M}^S$) präsentiert. Zur besseren Unterscheidung haben wir hier neue Namen für die Komponenten und ihre Modifikationen eingeführt.

Damit sind $\mathcal{M}^\Phi$ und $\mathcal{M}^S$ korrespondierende Modifikationen bezüglich Merge^Φ und Merge^S.

□

4.6 Fehler in der Systemumgebung

Mit dieser Arbeit wollen wir - wie bereits in Kapitel 1 motiviert - Methoden dafür anbieten, wie Systeme schrittweise entwickelt werden können, ausgehend von einer Idealisierung des Systems bis zu einer realen Implementierung, also auch unter Berücksichtigung

möglicher, zu erwartender Fehler. Mit den in diesem Kapitel bisher vorgestellten Begriffen können wir Modifikationen an der Schnittstelle und dem Verhalten von Systemen durchführen. Unberücksichtigt blieb dabei bislang die Umgebung der Systeme.

Fehler können auch in der Systemumgebung auftreten: Zu viele oder zu wenige, zu schnelle oder zu langsame bzw. inkonsistente Eingaben sind Beispiele, die es einem System unmöglich machen können, seine Funktionalität zu erbringen. Derartige Fälle lassen sich nicht durch eine Modifikation der Schnittstelle oder des Systemverhaltens darstellen.

Aufgrund der Linkstotalität der Verhaltensrelationen scheint ein Begriff des *externen Fehlers* nicht sinnvoll zu sein: Für jede Eingabe an ein System muss eine Ausgabe definiert sein. Die Umgebung kann also keine irgendwie gearteten *falschen* Nachrichtenströme an ein System schicken. Bei Black-Box Spezifikationen wird die Linkstotalität oft durch die Verwendung des *Assumption/Guarantee*-Formates erreicht, bei Transitionssystemen ist die Linkstotalität automatisch sichergestellt. Auch die Komposition mit Modifikationskomponenten erhält die Linkstotalität.

Dennoch kann ein Teil der Ausgaben unerwünscht sein, obwohl er in der verhaltensbeschreibenden Relation vorkommt. Existieren nämlich Umgebungsannahmen, die genau die Eingaben ausschließen, die zu den unerwünschten Ausgaben führen, wird das System diese Ausgaben nie zeigen. Wird eine Komponente beispielsweise in einem Kontext verwendet, der bestimmte Eingaben für das System nie produzieren wird, darf die Reaktion dieser Komponente auf diese Eingaben sogar beliebig sein - ihre Spezifikation wird damit nicht gefährdet, wenn sie nur Aussagen macht über die Reaktion auf eine Teilmenge der erlaubten Eingaben.

Diese *Umgebungsannahmen* sind damit die wesentliche Voraussetzung, um über Fehler der Systemumgebung sprechen zu können. Die Systemumgebung kann nur Fehler machen, wenn gewisse Anforderungen an sie gestellt werden. Diese Forderungen an die Umgebung können explizit oder implizit definiert sein. Wir behandeln im folgenden beide Fälle.

4.6.1 Explizite Umgebungsannahmen

Eine Umgebungsannahme nennen wir explizit, wenn sie beim Entwurf des Systems vom Entwickler identifiziert und dokumentiert wird. Dies geschieht, wenn erkennbar ist, dass das System nur in einem entsprechenden Kontext verwendet wird, und dass es seine Funktionalität tatsächlich nur unter diesen Bedingungen erbringen kann oder wenn es möglich ist, bei Kenntnis der Umgebungsannahmen eine (bezüglich geeigneter Kriterien) günstigere Implementierung zu wählen.

Eine Annahme bezüglich der Umgebung eines Systems ist eine Aussage über seine Eingabeströme, semantisch (im einfachsten Fall)[1] also eine Teilmenge von

$$type(i_1)^\omega \times \ldots \times type(i_n)^\omega$$

[1] Im allgemeinen Fall ist es nötig, diese Menge auch in Abhängigkeit von den Ausgaben des Systems selbst zu definieren, wie es in Kapitel 3.5 motiviert wurde. Wir beschränken uns hier auf den einfachen Fall.

Diese Menge enthält alle Eingabeströme, die die Umgebung an das System senden darf.

Die Teilmenge der zugelassenen Eingabeströme kann wieder - entsprechend den Beschreibungstechniken aus den Kapiteln 3.5 und 3.6 - auf zwei Arten beschrieben werden:

- Eine Black-Box Spezifikation der Umgebungsannahme wird durch eine Formel $A(i_1, \ldots, i_n)$[2] beschrieben, die die Namen der Eingabeströme (und möglicherweise auch der Ausgabeströme) als freie Variable enthält. Diese Annahme kann unabhängig von der Beschreibungstechnik für das System selbst formuliert werden, insbesondere also auch für Transitionssysteme. Wurde das System im Format einer A/G-Spezifikation definiert, so ist die Annahme A darin bereits üblicherweise definiert.

- Die Menge der gestatteten Eingabeströme kann auch beschrieben werden durch eine fiktive Zustandsmaschine $\mathcal{A}$, die genau diese Menge von Strömen erzeugen kann. Sie kann im einfachen Fall keine Eingaben besitzen und die Ausgaben spontan erzeugen oder die Ausgaben des Systems als Eingabe erhalten, um Rückschlüsse über den Zustand des Systems ziehen zu können.

  Zum Ableiten von Eigenschaften des Systems unter dieser Annahme betrachtet man die Komposition des System mit der abstrakten Umgebungsmaschine.

Der Begriff des Umgebungsfehlers läßt sich relativ zur Umgebungsannahme leicht definieren: Ein Fehler im Verhalten der Umgebung tritt auf, wenn die Umgebung Nachrichtenströme an das System schickt, die *nicht* die Annahme erfüllen.

Diese Menge von Strömen, die die Annahme nicht erfüllen, können wir unterteilen in zwei Teilmengen: einerseits die Fehler, die wir für plausibel halten und für deren Behandlung wir Maßnahmen im System ergreifen werden, und andererseits die Fehler der Umgebung, bei denen wir davon ausgehen, dass sie nicht eintreten werden und die wir auch nicht weiter berücksichtigen wollen. Um eine Trennung dieser beiden Menge explizit zu machen, wollen wir die Umgebungsannahmen derart abschwächen, dass der letztere Teil noch immer ausgeschlossen bleibt, aber das Auftreten spezifizierter Fehler möglich wird. Es stellt sich somit die Frage, wie eine konkrete Abschwächung für ein konkretes System beschrieben werden kann.

Wir schlagen hier zwei Varianten vor, die sich an der Beschreibung der Umgebungsannahme orientieren:

- Ist eine Umgebungsannahme durch eine Black-Box Spezifikation A beschrieben, so steht uns mit den *Modifikationen* bereits ein Formalismus bereit, mit dem Abschwächungen beschrieben werden können. Beschränken wir uns also auf eine Modifikation $\mathcal{M}^\Phi = (\mathsf{true}, \Phi^F)$, so kann damit ein Umgebungsfehler spezifiziert werden.

- Ist die Annahme an die Umgebung durch eine Umgebungskomponente $\mathcal{A}$ definiert, so können wir diese durch eine Modifikation für Transitionssysteme $\mathcal{M}^A = (\varnothing, F)$ spezifizieren, die der Umgebung damit zusätzliche Freiheiten in ihrem Verhalten zugesteht.

[2] Im allgemeinen Fall hätte diese Formel die Form $A(i_1, \ldots, i_n, o_1, \ldots, o_m)$.

Wir haben uns in beiden Fällen beschränkt auf eine reine Abschwächung des Systemverhaltens, lassen also weder eine Verstärkung Φ^E noch das Entfernen von Transitionen in $\mathcal{A}$ über ein nichtleeres E zu. Wir interessieren uns im Rahmen unseres idealen Entwicklungsprozesses für die Verbesserung von Systemen, was im Kontext von Umgebungsfehlern eine Erhöhung der Robustheit bedeutet. Ein System soll also trotz auftretender Umgebungsfehler sinnvolles Verhalten aufweisen. Im Kapitel 5.6 werden wir die Anpassung von Systemen an Umgebungsfehler diskutieren.

Selbstverständlich kann es möglich sein, auch die Annahmen an das Umgebungsverhalten verstärken zu wollen. Werden im Rahmen einer Modifikation die Anforderungen an ein System verstärkt, kann es notwendig sein, auch die Umgebungsannahmen entsprechend zu verstärken, ohne die ein System die verlangte, neu aufgenommene Forderung nicht zu erfüllen vermag. Die Umgebung kann dargestellt sein durch weitere Komponenten, die dann wiederum geeignet modifiziert werden müssen. In diesem Abschnitt fokussieren wir aber auf *Umgebungsfehler*, interpretiert als eine potentielle Möglichkeit der Umgebung, vom erwarteten Verhalten abzuweichen. Hierfür ist die Beschränkung auf abschwächende Modifikationen ausreichend.

4.6.2 Implizite Umgebungsannahmen

Ist ein System durch ein Transitionssystem spezifiziert, so kann eine Umgebungsannahme explizit definiert sein wie in Abschnitt 4.6.1. Aber auch ohne direkte Formulierung kann eine implizite Umgebungsannahme in den Entwurf des Transitionssystems eingeflossen sein.

Wir zielen in dieser Arbeit in erster Linie auf die Modellierung reaktiver Systeme ab. Ein System ist nur dann reaktiv, wenn es jederzeit bereit ist, weitere Nachrichten auf den Eingaben zu empfangen und darauf zu reagieren. Gerät das System in einen sogenannten *Fangzustand*, in dem zwar noch unverarbeitete Nachrichten vorliegen, aber keine Transition mehr schaltbereit ist, ist die Reaktivität des Systems nicht mehr gewährleistet.

Für auftretende Fangzustände kann es zwei Gründe geben:

- Der Entwurf des Transitionssystems ist fehlerhaft, und es wurden geeignete Transitionen nicht in das System aufgenommen, die für eine volle Reaktivität notwendig sind.
- Bei dem Entwurf des Systems wurden implizite Annahmen an die Umgebung gemacht, die Eingaben ausschließen, die zu einem Fangzustand führen würden. So können beispielsweise Kontrollzustände in ein Transitionssystem aufgenommen werden, in dem nicht von einem bestimmten Kanal gelesen werden kann, da der Entwickler annimmt, dass in einem solchen Systemzustand immer andere Nachrichten auf anderen Kanälen vorliegen werden, die eine Weiterverarbeitung (und damit das Lesen der zunächst nicht konsumierbaren Nachricht) in reaktiver Weise möglich machen. Diese Annahme wurde unter Umständen nicht explizit dokumentiert, ist aber sozusagen „versteckt“ im System enthalten.

Wir gehen im folgenden vom letztgenannten Grund aus und interpretieren mögliche

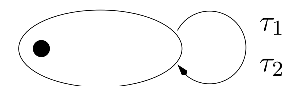


Abbildung 4.7: Transitionssystem des Puffers

Fangzustände als implizite Umgebungsannahme. Diese fordert von den von der Umgebung produzierten Eingaben, dass sie vollständig gelesen werden können, also das System nicht in einen Fangzustand geraten lassen.

Definition 4.14 *Implizite Umgebungsannahme*

Die implizite Umgebungsannahme $\mathcal{A}$ eines Transitionssystems $\mathcal{S}$ wird definiert als die Menge aller Eingabeströme, die vollständig gelesen werden können, für die das System also in keinem Ablauf in einen Fangzustand gerät. Die Menge wird ausgedrückt durch Belegungen, die (unter anderem) den Eingabekanälen $i \in I$ Ströme zuweisen:[3]

$$\mathcal{A}(\mathcal{S}) \quad \stackrel{df}{=} \quad \{\alpha \mid \forall\, \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \forall\, i \in I \bullet \xi.\infty(i) = \xi.\infty(i^\circ) \wedge \alpha \stackrel{I}{=} \xi.\infty\}$$

□

Wir illustrieren diesen Ansatz, in dem wir den Puffer aus Beispiel 3.4 als Transitionssystem spezifizieren und daraus eine implizite Umgebungsannahme ableiten.

Beispiel 4.12 Ableitung einer Umgebungsannahme

Der Puffer kann als Transitionssystem $\mathcal{S} = (I, O, A, \Upsilon, T)$ folgendermaßen definiert werden. Die Variablenmenge A wird definiert zu $\{i^\circ, q\}$. Der gelesene Teil von i wird damit standardmäßig ergänzt. Die Variable q vom Typ $type(q) = D^\omega$ verwenden wir als eigentlichen Puffer, in dem die Daten gespeichert werden. Die Konstante N muss nicht in A aufgenommen werden, da wir sie als globale Größe voraussetzen. Da wir keine Kontrollzustände benötigen, ist dafür auch keine Variable nötig. Die Initialbedingung definieren wir zu

$$\Upsilon \stackrel{df}{=} q = \langle\rangle \wedge i^\circ = \langle\rangle \wedge o = \langle\rangle$$

Dies besagt, dass der Puffer initial leer ist und der Puffer weder Eingaben gelesen noch Nachrichten gesendet hat.

Wir benötigen zwei Transitionen τ_1 und τ_2 in T, jeweils für eine Nachricht vom Typ D und für eine Anfrage R. Das sehr einfache STD ist in Abbildung 4.7 dargestellt. Die beiden Transitionen spezifizieren wir tabellarisch:

Name	*Pre*	*Input*	*Output*	*Post*
τ_1	$\#q < N$	$i?d$	–	$q' = q \frown \langle d \rangle$
τ_2	$\#q > 0$	$i?R$	$o!ft.q$	$q' = rt.q$

Die Transition τ_1 liest ein Datum d vom Eingabekanal und hängt dieses an q an, aber nur, wenn die Länge von q die Kapazitätsgrenze von N noch nicht erreicht hat. Die Transition τ_2 reagiert auf den Empfang einer Anfrage R durch die Ausgabe des ersten Elementes in q und

[3] Der Ausdruck $\xi.\infty$ beschreibt den fiktiven Zustand, den ein System nach einem unendlichen Ablauf „erreicht". Er wird definiert in Definition 3.7.

dem Entfernen dieses Elements aus q, aber nur, wenn wirklich Daten in q gespeichert sind. Diese Vorbedingung stellt sicher, dass der Ausdruck $ft.q$ definiert ist.

Aus diesem Transitionssystem wollen wir nun die darin enthaltene implizite Umgebungsannahme ableiten. Wir wollen dazu die Eingabeströme betrachten, bei denen das System nicht in einen blockierenden Zustand gerät. Hierfür charakterisieren wir zunächst die Umkehrung, also die Eingabeströme, bei denen das System hängenbleibt. Für unseren Puffer sind diese Fälle relativ leicht zu charakterisieren, da es nur einen Eingabestrom und einen Kontrollzustand gibt. Wir unterscheiden dazu nach der nächsten zu lesenden Nachricht m vom Eingabestrom, für die also gilt:

$$i^\circ \frown \langle m \rangle \sqsubseteq i$$

Entsprechend der Typinformation über den Kanal i sind zwei Fälle möglich:

- Die nächste zu lesende Nachricht ist ein Element d vom Typ D. Diese muss von Transition τ_1 gelesen werden, die aber nur schaltbereit ist, wenn $\#q < N$. Der Puffer blockiert also, wenn gilt

  $$\Phi_1 \quad \equiv \quad \exists\, d \bullet i^\circ \frown \langle d \rangle \sqsubseteq i \land \#q \geq N$$

- Als nächste Nachricht muss die Nachricht R konsumiert werden. Die zuständige Transition τ_2 kann dies nur durchführen, wenn die Vorbedingung $\#q > 0$ erfüllt ist. Ein blockierender Zustand ist also beschrieben durch

  $$\Phi_2 \quad \equiv \quad i^\circ \frown \langle R \rangle \sqsubseteq i \land \#q \leq 0$$

Ein Eingabestrom i kann nur dann vollständig gelesen werden, wenn die beiden obigen Fälle nie eintreten, wenn also

$$\forall\, i^\circ \bullet \neg\, \Phi_1 \land \neg\, \Phi_2$$

erfüllt ist. Wir leiten daraus nun eine Anforderung an den Eingabestrom ab.

Durch Expansion der Negation ergibt sich

$$\begin{array}{ll} \forall\, i^\circ, d \bullet & i^\circ \frown \langle d \rangle \sqsubseteq i \Rightarrow \#q < N \quad \land \\ & i^\circ \frown \langle R \rangle \sqsubseteq i \Rightarrow \#q > 0 \end{array}$$

Wir müssen nun eine Invariante des Puffers verwenden, die den Zusammenhang zwischen der Länge des Puffers und dem Eingabestrom beschreibt. Sie kann mit den Techniken aus Abschnitt 3.8.2 leicht bewiesen werden.

$$\#q = \#D \circledS i^\circ - \#\{R\} \circledS i^\circ$$

Damit ergibt sich durch Ersetzung

$$\begin{array}{ll} \forall\, i^\circ, d \bullet & i^\circ \frown \langle d \rangle \sqsubseteq i \Rightarrow \#D \circledS i^\circ - \#\{R\} \circledS i^\circ < N \quad \land \\ & i^\circ \frown \langle R \rangle \sqsubseteq i \Rightarrow \#D \circledS i^\circ - \#\{R\} \circledS i^\circ > 0 \end{array}$$

und daraus unter Ausnutzung der Eigenschaften der Operators $\circledS$

$$\begin{array}{lll} \forall\, i^\circ, d \bullet & i^\circ \frown \langle d \rangle \sqsubseteq i \Rightarrow & \#D \circledS (i^\circ \frown \langle d \rangle) - 1 \\ & & -\ \#\{R\} \circledS (i^\circ \frown \langle d \rangle) < N \quad \land \\ & i^\circ \frown \langle R \rangle \sqsubseteq i \Rightarrow & \#D \circledS (i^\circ \frown \langle R \rangle \\ & & -\ (\#\{R\} \circledS (i^\circ \frown \langle R \rangle) - 1) > 0 \end{array}$$

woraus wir trivialerweise schließen können auf

$$\begin{array}{ll} \forall\, i^\circ, d \bullet & i^\circ \frown \langle d \rangle \sqsubseteq i \Rightarrow \#D\circledS(i^\circ \frown \langle d \rangle) - \#\{R\}\circledS(i^\circ \frown \langle d \rangle) \leq N \quad \wedge \\ & i^\circ \frown \langle R \rangle \sqsubseteq i \Rightarrow \#D\circledS(i^\circ \frown \langle R \rangle) - \#\{R\}\circledS(i^\circ \frown \langle R \rangle) \geq 0 \end{array}$$

Damit ist für jeden Präfix i', ob er nun mit einer Nachricht d oder R endet, gezeigt, dass

$$\Phi \equiv \forall\, i' \sqsubseteq i \bullet 0 \leq \#D\circledS i' - \#\{\circledR\}\circledS i' \leq N$$

was genau der Umgebungsannahme der Black-Box Spezifikation des Puffers entspricht. Beim Entwurf des Transitionssystems ist diese Annahme implizit mit eingeflossen. Die Menge $\{\alpha \mid \Phi\}$ enthält die in Definition 4.14 beschriebene Menge von Belegungen. □

Sind Fangzustände versehentlich, zum Beispiel durch einen Entwurfsfehler in einem System enthalten, so kann eine explizite Formulierung der darin enthalten Umgebungsannahme helfen, diesen Entwurfsfehler aufzudecken: Es ließe sich erkennen, dass die Umgebungsannahme nicht intendiert und nicht realistisch ist, da sie der Umgebung zuviele Eingaben an das System verbietet.

Fangzustände können auch bewußt in ein System aufgenommen werden, da ein System in bestimmten Zuständen tatsächlich stehen bleiben soll. Der hier vorgestellte Ansatz der impliziten Umgebungsannahmen scheint dann nicht anwendbar zu sein, denn es würde von der Umgebung gefordert werden, nur Eingaben zu senden, die nicht zu diesen Zuständen führen. Dieses Problem wird vermieden, indem zu den gewollten Haltezuständen Transitionen ergänzt werden, die alle Nachrichten von allen Kanälen konsumieren können ohne dabei eine Reaktion durch Ausgaben sichtbar zu machen. Befolgt man dies als Entwurfsdisziplin, lassen sich die verbleibenden „wahren“ impliziten Umgebungsannahmen wieder sinnvoll definieren.

Die Menge aller Eingaben, die die Annahme $\mathcal{A}(\mathcal{S})$ verletzen, stellen Fehler der Umgebung dar, wenn diese dennoch an das System geschickt werden. Eine sinnvolle Weiterentwicklung des Systems besteht, wie bereits im vorausgehenden Abschnitt erläutert, in der *Erhöhung der Robustheit*. Dies wollen und können wir meist nicht für *alle* denkbaren externen Fehler tun, sondern nur für eine Teilmenge, für die eine passende Reaktion zu definieren ist. Wir differenzieren also auch hier wieder zwischen der Menge der berücksichtigten Fehler, die erwartet, beschrieben und behandelt werden und der Menge der nicht-berücksichtigten Fehler. Von letzteren nehmen wir an, dass sie nie auftreten, und ignorieren damit den Fall, falls dies dennoch passieren sollte.

Die Teilmenge der Fehler, die zu berücksichtigen sind, muss durch eine geeignete Beschreibungstechnik definiert werden können. Dies ist auf der Basis eines Transitionssystems $\mathcal{S} = (I, O, A, \Upsilon, T)$ möglich, indem wir mit einer Formel Φ_{trap} mit freien Variablen aus $I \cup O \cup A$ eine Teilmenge von Zuständen des Systems beschreiben. Diese Menge charakterisiert die Eingaben, die zu diesen Zuständen führen durch

$$\mathit{LeadsTo}(\Phi_{trap}) \overset{df}{=} \{\alpha \mid \exists\, \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \exists\, t \bullet \xi.t \models \Phi_{trap} \wedge \alpha \overset{I}{=} \xi.t\}$$

Werden Transitionen dem System hinzugefügt, so dass alle Zustände, für die Φ_{trap} gilt, keine Fangzustände mehr sind, so entspricht dies einer Abschwächung der Umgebungsannahme des Systems. Das modifizierte System bleibt dann in einer Umgebung reaktiv, die Eingaben aus $\mathcal{A}(\mathcal{S}) \cup \mathit{LeadsTo}(\Phi_{trap})$ erzeugt. In Abschnitt 5.6 werden wir

die entsprechenden Systemmodifikationen beschreiben und mit dem Puffer als Beispiel illustrieren.

Alle in unserem Systemmodell spezifizierten Systeme weisen eine Umgebungsannahme auf, die bislang nicht angesprochen wurde: Wir nehmen von allen Nachrichten a, die über einen Kanal c empfangen werden, an, dass sie zum richtigen Nachrichtentyp gehören, also $a \in \mathit{type}(c)$. Soll modelliert werden, dass ein System unerwartete Nachrichten empfängt, so ist dies durch eine Kombination bisher dargestellter Techniken möglich: Zunächst wird durch eine Schnittstellenmodifikation der Kanal c um die neuen Nachrichten zu c' ergänzt und die Umgebungsannahme um $\forall\, t \bullet c.t \in \mathit{type}(c)$ erweitert. Der Fehler der Umgebung ist dann durch eine Abschwächung der Umgebungsannahme modellierbar.

4.7 Zusammenfassung und Diskussion

Wir haben in diesem Kapitel das Konzept der Modifikationen eingeführt. Mit diesen können beliebige Veränderungen sowohl der Schnittstelle als auch des Verhaltens eines Systems dargestellt werden. Mit Modifikationen kann prinzipiell jedes System in jedes andere verwandelt werden. In der Praxis wird man aber eher an relativ kleinen Änderungen interessiert sein, mit denen Fehler als Abweichungen eines Ist von einem Soll repräsentiert werden. Die Modifikationen wurden im Rahmen von vier Beschreibungstechniken eingeführt: Als Modifikationen der Relationen, der Black-Box Eigenschaften und der Transitionssysteme und schließlich als Modifikationskomponenten. Damit ermöglichen sie die Formalisierung von Fehlern auf verschiedenen Abstraktionsstufen.

Auch die sogenannte *Schnittstellenverfeinerung*, die uns im Systemmodell von Focus zur Verfügung steht, erlaubt die Modifikation von Schnittstellen: Eine abstrakte Darstellung einer Kommunikationsgeschichte kann durch Repräsentations- und Abstraktionsprädikate zu einer konkreteren Darstellung in Beziehung gesetzt werden. Diese Art der Modifikation ist aber einerseits zu mächtig, da mit ihr auch eine Verhaltensveränderung modelliert werden kann. Unser Ziel war eine Trennung der Modifikation von Verhalten und Schnittstelle, um die Komplexität eines Einzelschrittes überschaubar zu halten. Andererseits sind Schnittstellenverfeinerungen durch ihre Forderungen an die Repräsentations- und Abstraktionsprädikate zu eingeschränkt, um alle Modifikationen beschreiben zu können. Modifikationen sind im allgemeinen keine Verfeinerungen, sondern erlauben eine wirkliche Veränderung eines Systems.

Bei der Definition der Verhaltensmodifikationen wird, entsprechend Abbildung 4.2, in den drei Formalismen der Relationen, Eigenschaften und Transitionen ein Teil des Verhaltens entfernt und ein anderer Teil hinzugefügt. Durch die Klammerung ist eine Reihenfolge dieser Operationen definiert: Das Entfernen wird zuerst durchgeführt, das Hinzufügen danach. Diese Reihenfolge ist keinesfalls zwingend, und hätte auch anders gewählt werden können. Unsere Wahl erweist sich dennoch günstiger, wie am Beispiel 5.5 erkennbar wird: Würde man zunächst die Transitionen ergänzen, die c auf 1 bzw. 2 setzen, so dürften diese danach nicht mehr entfernt werden. Die Formulierung von E_1 wäre aufwendiger.

Mit Hilfe der formalisierten Modifikationen waren wir in der Lage, die sonst nur informell charakterisierten Begriffen für *fault*, *error* und *failure* mit einer formalen Definition zu präzisieren. Im nächsten Kapitel werden wir diese Begriffe nutzen können, um beispielsweise die Korrektheit von fehlererkennenden Mechanismen auszudrücken.

Auch den Begriff der *Fehlertoleranz* können wir mit Hilfe der Modifikationen präzise definieren. Fehlertoleranz wird meist als die Fähigkeit eines Systems verstanden, unter dem Einfluß von Fehlern dennoch eine akzeptable Leistung zu erbringen. Durch Modifikationen können wir sowohl die Fehler als auch die Veränderung im Verhalten explizit ausdrücken. In einer anderen Sichtweise von Fehlertoleranz wird sie als die Abschwächung von Fehlereinflüssen interpretiert: Auftretende Fehler in einer Teilkomponente oder der Systemumgebung werden zumindest teilweise toleriert und zeigen nur noch eine abgeschwächte Wirkung auf das Gesamtverhalten. Wir haben diese Sichtweise skizziert, und eine Vereinfachung aufgezeigt, die auf quantitative Bewertungen der Schwere von Fehlern verzichten kann.

In weiteren Untersuchungen wurden formale Eigenschaften der Modifikationen behandelt, die in einem Entwicklungsprozeß fehlerbehafteter Systeme eingesetzt werden können. Wie in Kapitel 3.10 beschrieben, resultiert ein (idealisierter) Entwicklungsprozeß in einem Entwicklungsbaum mit einer abstrakten Spezifikation als Wurzel, einer (iterierten) Verzweigung in Spezifikationen von Teilkomponenten, und schließlich einem Übergang zu Transitionssystemen als Blättern. Werden an einer Stelle in diesem Baum Modifikationen durchgeführt, ist man an einer Anpassung des Baumes interessiert, so dass ein modifizierter, aber wieder korrekter Entwicklungsbaum vorliegt. Mit den Abschnitten über die Fehlerfortpflanzung sind die formalen Grundlagen für die Modifikationen innerhalb eines Beschreibungsformalismus bereits dargestellt. Für die Anpassung des Übergangs von Transitionssystemen zu Black-Box Eigenschaften wird im nächsten Kapitel eine methodische Unterstützung angeboten.

Schliesslich haben wir die Thematik der externen Fehler behandelt. Aufgrund des Schnittstellenbegriffs unseres Systemmodells können wir klar zwischen internen und externen Fehlern trennen. Wir haben sowohl explizite als auch implizite Fehlerannahmen diskutiert, mit den externe Fehler definiert werden können. Den Umgang mit externen Fehlern werden wir in Abschnitt 5.6 behandeln.

Kapitel 5

Methodischer Umgang mit Fehlern

In den vorangegangenen Kapiteln wurden die formalen Grundlagen vorgestellt, mit denen verteilte reaktive Systeme und Modifikationen sowohl der Schnittstellen als auch des Verhaltens dieser Systeme dargestellt werden können. Damit steht uns zunächst nur eine *Sprache* zur Verfügung, mit der Fehler und fehlerbehaftete Systeme ausgedrückt werden können. Um diese Sprache aber auch gewinnbringend zur Entwicklung fehlertoleranter Systeme einsetzbar zu machen, werden wir in diesem Kapitel ihre *Verwendung* darstellen. Dieses Kapitel erläutert dazu, wie man beispielsweise die Fehlererkennung zu gegebenen Fehlern in ein System integriert, wie man ein System derart erweitert, dass es Fehlermeldungen erzeugen kann und mit welchen Techniken fehlerkorrigierende Mechanismen ergänzt werden können.

Dazu geben wir zunächst Beispiele für die Formalisierung typischer Fehlermodelle an. Sollen die Zusammenhänge von Fehlern in verschiedenen Komponenten ausgedrückt werden, können sogenannte virtuelle Komponenten und Orakel verwendet werden, die wir in Abschnitt 5.2 beschreiben. Die Fehlererkennung kann durch Prädikate realisiert werden, deren Eigenschaften wir in Abschnitt 5.3 charakterisieren. Das Einführen von Fehlermeldungen beschreiben wir in Abschnitt 5.4. In Abschnitt 5.5 geben wir zwei Techniken an, wie Mechanismen zur Fehlerkorrektur nachträglich in ein System integriert werden, sowohl basierend auf Transitionssystemen als auch auf Modifikationskomponenten. Daraufhin diskutieren wir die Möglichkeiten, ein bestehendes System nachträglich robuster gegenüber Fehlern in der Systemumgebung zu machen. In Abschnitt 5.7 wird schließlich aufgezeigt, wie Beweise, die für ein fehlerfreies System geführt wurden, an die fehlerbehaftete Version angepasst werden können.

5.1 Beispiele für Fehlermodelle

Der Umgang mit Fehlern erfordert eine explizite Formulierung der Fehlerklassen, mit deren Auftreten man rechnet. Zur Illustration der Ausdrucksmächtigkeit unserer Beschreibungstechniken geben wir in diesem Abschnitt die Formalisierung einiger Beispiele

an, die in der Literatur als typische Fehlerklassen identifiziert werden [23, 27].

Wir modellieren die folgenden Fehlerklassen sowohl im Formalismus der Black-Box Spezifikationen als auch mit Transitionssystemen. Sei dazu eine Spezifikation Φ bzw. ein Transitionssystem $\mathcal{S}$ mit $\mathcal{S} = (I, O, A, \Upsilon, T)$ gegeben, an das wir (bis auf die Annahme der Existenz von Ausgabekanälen) keine weiteren Anforderungen stellen. Die folgenden als Modifikationen dargestellten Fehlermodelle können also in Verbindung mit beliebigen Systemen verwendet werden.

5.1.1 Crash failure

Bei diesem Fehlermodell kann ein System spontan jegliche Aktivitäten einstellen und somit keine Nachrichten mehr auf seinen Ausgabekanälen ausgeben.

Im Rahmen der Black-Box Spezifikationen stellen wir dies durch eine Modifikation $\mathcal{M} = (\mathsf{true}, \Phi_F)$ dar, wobei Φ_F partielle Ausgabeströme ergänzt. Wir gehen hier vereinfachend von nur einem Ein- und Ausgabekanal aus. Ein Strompaar (i, o) repräsentiert ein zulässiges Verhalten des fehlerhaften Systems mit potentiellem crash failure-Fehlermodell, wenn die Verlängerung o' von o ein Normalverhalten darstellt:

$$\Phi_F \stackrel{df}{=} \exists\, o' \bullet o \sqsubseteq o' \;\wedge\; \Phi[o'/o]$$

Im Formalismus der Transitionssysteme modellieren wir dieses Fehlermodell durch die Hinzunahme einer neuen, in A noch nicht enthaltenen booleschen Variable, die wir mit *stop* bezeichnen. Die Hinzunahme kann entsprechend Kapitel 4.2.3 zunächst verhaltensneutral erfolgen.

Der Wahrheitswert der Variable gibt an, ob das System schon seinen Stoppzustand erreicht hat. In einem solchen Fall darf keine der Transitionen T schaltbereit sein. Daher sind alle Transitionen aus dem System zu entfernen, die von einem Zustand α ausgehen, in dem $\alpha.stop = \mathsf{true}$ gilt. Dies wird durch folgende Definition von E aus der Modifikation $\mathcal{M} = (E, F)$ erreicht.

$$E \stackrel{df}{=} \{(\alpha, \beta) \mid \alpha.stop = \mathsf{true}\} \;\cup\; \{(\alpha, \beta) \mid \alpha.stop \neq \beta.stop\}$$

Wie in Abschnitt 4.2.3 beschrieben, entfernen wir standardmäßig mit diesem E zusätzlich alle Transitionen, die den Wert der neuen Variablen ändern könnten. Nun ergänzen wir mit Hilfe von F eine stets schaltbereite Transition τ_{stop}, die jederzeit in der Lage ist, den Wert von *stop* auf true zu setzen und damit das System zu blockieren:

$F \stackrel{df}{=}$

Name	*Pre*	*Input*	*Output*	*Post*
τ_{stop}	true	–	–	$stop' = \mathsf{true}$

Da keine Transition im System enthalten ist, die *stop* wieder auf false setzen kann, bleibt ein einmal angehaltenes System immer gestoppt. Durch eine stärkere Vorbedingung *Pre* von τ_{stop} kann der Ausfall des Systems von anderen Systemparametern abhängig gemacht werden.

Eine Veränderung der Initialbedingung Υ ist nicht nötig, der initiale Wert von *stop* ist also frei wählbar. Damit ist es möglich, dass das durch $\mathcal{S} \triangle \mathcal{M}$ definierte System gleich von Anbeginn seiner Ausführung versagt.

5.1.2 Fail-stop failure

Wie beim crash failure-Modell bleibt bei diesem Fehlermodell das System spontan stehen. Im Unterschied zu crash-failure sendet es aber noch ein Signal an seine Umgebung, welches das Versagen leicht und unmittelbar erkennbar macht.

Aufgrund der Ähnlichkeit der beiden Modelle unterscheiden sich auch die Formalisierungen nur minimal. Wir gehen von einem existierenden Kanal f aus, auf dem Fehlermeldungen *fail* ausgegeben werden können, also $\mathit{fail} \in \mathit{type}(f)$. Unter Umständen ist dieser Kanal entsprechend Abschnitt 4.1.1 mit Hilfe einer (verhaltensneutralen) Schnittstellenmodifikation zu ergänzen.

Wir können die Modifikation (true, Φ_F) der Black-Box Spezifikation so anpassen, dass bei einem auftretenden Versagen des Systems auch wirklich eine Fehlermeldung auf f ausgegeben wird. Der Strom o muss in diesem Fall ein echter Präfix von o' sein, da die Fehlermeldung nur genau dann ausgegeben werden soll, wenn der Fehler wirklich aufgetreten ist:

$$\Phi'_F \stackrel{df}{=} \exists\, o' \bullet o \sqsubset o' \;\wedge\; \Phi[o'/o] \;\wedge\; \mathit{lt}.f = \mathit{fail}$$

Als entsprechende Modifikation des Transitionssystems passen wir F so an, dass bei Ausfall des Systems die Fehlermeldung sofort erzeugt wird:

$F' \stackrel{df}{=}$

Name	*Pre*	*Input*	*Output*	*Post*
τ'_{stop}	true	–	$f!\mathit{fail}$	$\mathit{stop}' = \mathsf{true}$

Die Modifikationen $\mathcal{M} = (\mathsf{true}, \Phi'_F)$ bzw. $\mathcal{M} = (E, F')$ beschreiben damit das *fail-stop failure*-Modell für jedes System mit $f \in O$.

5.1.3 Omission failure

Mit *omission failure* bezeichnet man Verluste in der Ausgabe eines Systems. Wir beschreiben dieses Fehlermodell bezogen auf einen Ausgabekanal $o \in O$ des Systems $\mathcal{S}$. Die Verluste treten vollkommen nichtdeterministisch auf, d.h. alle Varianten zwischen den Extremen – Verlust *aller* Nachrichten einerseits und überhaupt *keinem* Verlust andererseits – sind möglich. Das Normalverhalten bleibt dabei also ebenso möglich, so dass wir Modifikationen mit $\Phi_E = \mathsf{true}$ bzw. $E = \varnothing$ wählen.

Für die neuen Paare (i, o) im Fehlerverhalten soll gelten, dass der Strom o eine beliebige Zahl von Nachrichten nicht mehr enthält. Reichert man o wieder an zu o', so stellt o' ein Normalverhalten dar. Formal definieren wir dieses Fehlermodell durch

$$\Phi_F \stackrel{df}{=} \exists\, o' \bullet (o, o') \in \mathit{EXTEND}(\mathit{type}(o)) \;\wedge\; \Phi[o'/o]$$

Die Relation $EXTEND(M)$ enthält alle Strompaare (x, y) vom Typ M, in denen y durch eine Anreicherung von x um Nachrichten aus M entstehen kann:

$$y \in M^\omega \quad \Rightarrow \quad (\langle\rangle, y) \in EXTEND(M)$$

$$d \in M \wedge x, y \in M^\omega \wedge z \in M^* \wedge (x, y) \in EXTEND(M)$$
$$\Rightarrow \quad (\langle d\rangle \frown x, z \frown \langle d\rangle \frown y) \in EXTEND(M)$$

Für Transitionssysteme modellieren wir dieses Fehlermodell durch eine Ergänzung neuer Transitionen vermöge der Modifikation $\mathcal{M} = (\varnothing, F)$ mit

$$F \stackrel{df}{=} \{(\alpha, \beta) \mid \exists \gamma \bullet (\alpha, \gamma) \in T \;\wedge\; \beta.o = \alpha.o \;\wedge\; \beta \stackrel{V-\{o\}}{=} \gamma\}$$

Wir fügen also diejenigen Transitionen (α, β) hinzu, die den bestehenden in T sehr ähnlich sind und sich nur darin unterscheiden, dass sie den Ausgabestrom auf dem Kanal o unverändert lassen. Man ergänzt also zu jeder Transition, die eine Ausgabe auf o macht, eine weitere Transition, die sich ebenso verhält, aber die Ausgabe auf o nicht macht. Obige Formalisierung macht diese Intuition nicht unmittelbar deutlich. Hat man ein konkretes System beispielsweise tabellarisch beschrieben, lassen sich die zusätzlichen Transitionen anschaulicher beschreiben: Für jede Transition, die in der tabellarischen Darstellung in der Spalte *Output* einen Ausdruck der Form $o!expr$ hat, ergänzen wir dieselbe Transition, aber ohne diese Ausgabeanweisung.

5.1.4 Byzantine failure

Bei diesem Fehlermodell verhält sich das betroffene System auf völlig unvorhersehbare Weise. Da damit jedes Verhalten möglich wird, bezeichnen man dies auch als *chaotisches* Verhalten. Es ist sehr leicht zu formalisieren durch die Modifikationen $\mathcal{M} = (\mathsf{true}, \mathsf{true})$ bzw. $\mathcal{M} = (\varnothing, F)$, wobei F die Gesamtheit aller Transitionen enthält:

$$F \stackrel{df}{=} VAL \times VAL$$

Bei diesem Fehlermodell liegt keinerlei Regelmäßigkeit im Verhalten vor, so dass sich daraus keine Eigenschaften mehr ableiten lassen. Es beschreibt damit das „schlimmste“ aller denkbaren Fehlermodelle.

Die Beliebigkeit des Verhalten ist etwas eingeschränkt, da in einem realen System natürlich nicht alle Paare (i, o) bzw. alle Transitionen aus F wirklich auftreten können. So muss beispielsweise die Kausalität und die Monotonie auf den Kanälen (entsprechend Abschnitt 3.4) erhalten bleiben. Auch durch einen Fehler kann ein System nicht an Informationen gelangen, die noch nicht verfügbar sind. Alle Fehler, die in der Realität auftreten können, werden aber mit diesem Fehlermodell erfaßt.

Mit unseren Beschreibungstechniken sind wir in der Lage, für verschiedenartige Fehlermodelle eine Formalisierung anzugeben. Diese Modelle können in Untersuchungen verwendet werden, in denen das Verhalten von Systemen unter dem Einfluß von Fehlern analysiert wird, und es werden Fehlertoleranzaussagen möglich gemacht. Als Beispiel

sei das Alternating Bit Protokoll erwähnt. Dieses Protokoll ermöglicht eine Übertragung von Daten über Kanäle, die *omission failures* aufweisen, versagt aber, falls auf den Kanälen ein *crash failure* auftritt. Ein formaler Beweis dieser Aussage kann auf die hier vorformulierten Fehlermodelle zurückgreifen.

5.2 Formulierung von Fehlerannahmen

Wie wir gesehen haben, können wir die Fehler eines Systems durch Modifikationen darstellen. Die Ausdruckskraft der Modifikationen ist allerdings beschränkt, wenn spezielle, verfeinerte Fehlermodelle spezifiziert werden sollen, die Zusammenhänge von auftretenden Fehlern in *verschiedenen* Komponenten eines zusammengesetzten Systems beschreiben.

Dies kann mit dem Beispiel eines Triple-Modular-Redundancy Systems (beschrieben in Kapitel 2.2.2) veranschaulicht werden. Maskierende Fehlertoleranz kann im allgemeinen nur sichergestellt sein, wenn maximal eine der drei replizierten Komponenten versagt. Für einen formalen Nachweis der Fehlertoleranz muss diese Voraussetzung explizit repräsentiert werden. Dazu schlagen wir folgendes Verfahren vor:

- Das Versagen der drei replizierten Komponenten wird gekoppelt an einen Orakelstrom, den die Komponenten zusätzlich als Eingabe erhalten. Dieser Strom, typischerweise vom Typ $\{\mathsf{true}, \mathsf{false}\}$, gibt an, ob eine Komponente versagen darf.
- Die Voraussetzungen, die an das Auftreten der Versagen gestellt werden, können durch ein Prädikat über den Orakelstrom ausgedrückt werden oder durch eine virtuelle Komponente, die den Orakelstrom erzeugt.

In Abbildung 5.1 haben wir eine virtuelle Komponente V und die gestrichelt dargestellten Kanäle für die Orakelströme für ein Triple-Modular-Redundancy System visualisiert. Signalisiert die Komponente V immer nur maximal einer Komponente die „Erlaubnis" zum Versagen, kann für das Gesamtsystem maskierende Fehlertoleranz nachgewiesen werden. Die Komponente V wird nicht implementiert. Sie repräsentiert nur auf explizite Weise die globale Fehlerannahme, die damit in Beweisen verwendet werden kann.

Beispiel 5.1 Als Beispiel wählen wir das Triple-Modular-Redundancy System aus Abbildung 5.1. Eine Komponente sei dreifach repliziert als $\mathcal{S}_1, \mathcal{S}_2$ und $\mathcal{S}_3$, denen das *crash failure*-Fehlermodell zugewiesen sei. Der Voter ist durch Vergleich der Ausgaben der drei Komponenten in der Lage, den Ausfall einer Komponente an einer fehlenden Nachricht zu erkennen. Für das hier gewählte Fehlermodell kann das Gesamtsystem seine Leistung noch erbringen, wenn *maximal zwei* der drei Komponenten ausfallen. Diese Aussage wollen wir formal nachweisbar machen. Dazu führen wir die vorgeschlagenen beiden Schritte durch:

- Das Auftreten des Ausfalls wird kausal verknüpft mit einem entsprechenden Signal im Orakelstrom oc_i mit $i \in \{1, 2, 3\}$, der entsprechend Abschnitt 4.1 ergänzt wird. Das Fehlermodell aus Abschnitt 5.1.1 wird weiter angepasst durch eine Veränderung der Transition τ_{stop}. Diese Transition darf nur noch schalten, wenn sie auf oc_i den Wert true empfängt.

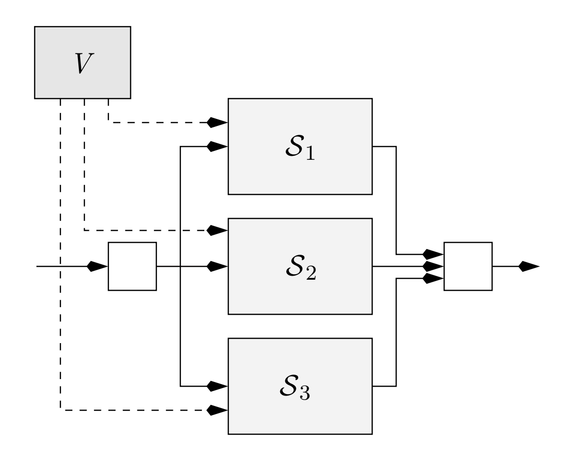


Abbildung 5.1: TMR-System mit virtueller Komponente

$F \stackrel{df}{=}$

Name	*Pre*	*Input*	*Output*	*Post*
τ_{stop}	–	$oc_i?\mathsf{true}$	–	$stop' = \mathsf{true}$

- Es darf nur auf maximal zwei der drei Kanäle oc_1, oc_2 und oc_3 die Nachricht true geschickt werden. Dies läßt sich durch eine Spezifikation der virtuellen Komponente V ausdrücken. Diese hat die Schnittstelle $(\varnothing, \{oc_1, oc_2, oc_3\})$ mit $type(oc_i) = \{\mathsf{true}\}$. Die „Erlaubnis" zum Versagen wird durch die Nachricht true dargestellt, ansonsten wird keine Nachricht verschickt.

  Das Verhalten läßt sich sowohl durch eine Black-Box Beschreibung durch

  $$oc_1 = \langle\rangle \vee oc_2 = \langle\rangle \vee oc_3 = \langle\rangle$$

  oder auch als einfaches Transitionssystem spezifizieren, in Abbildung 5.2 graphisch dargestellt. Die Transitionen sind definiert durch

Name	*Pre*	*Input*	*Output*	*Post*
τ	–	–	–	–
τ_1	–	–	$oc_1!\mathsf{true}$	–
τ_2	–	–	$oc_2!\mathsf{true}$	–
τ_3	–	–	$oc_3!\mathsf{true}$	–
τ_{12}	–	–	$oc_1!\mathsf{true}, oc_2!\mathsf{true}$	–
τ_{13}	–	–	$oc_1!\mathsf{true}, oc_3!\mathsf{true}$	–
τ_{23}	–	–	$oc_2!\mathsf{true}, oc_3!\mathsf{true}$	–

Da wir das ungezeitete Modell als Grundlage haben, besteht kein unmittelbarer zeitlicher Zusammenhang zwischen der Entscheidung von V, welche Transition ausgeführt wird (und welche Komponenten damit ausfallen dürfen) und dem tatsächlichen Ausfall. Wird eine Nachricht true von V auf einen Kanal oc_i geschrieben, wird sie von einer der drei Transitionen τ_{stop} zu einem späteren, aber sonst beliebigen Punkt in der Systemausführung gelesen. Aufgrund der Fairness in den Ausführungen ist sicher gestellt, dass eine vorhandene Nachrichten irgendwann gelesen wird, also ein Ausfall tatsächlich stattfinden wird. Die Transition τ beschreibt den Fall, dass keine der drei Komponenten einen Fehler aufweist.

Mit dieser Formalisierung des Gesamtsystems wird also die Annahme ausgedrückt, dass maximal zwei der drei Komponenten ausfallen. Die behauptete maskierende Fehlertoleranz des Systems kann damit nachgewiesen werden. □

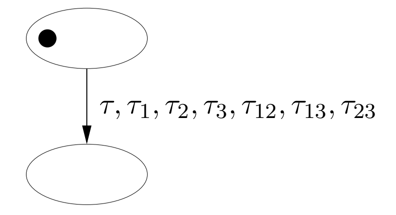


Abbildung 5.2: Transitionssystem der virtuellen Komponente

Orakelströme sind weiterhin dafür geeignet, Annahmen über *Häufigkeiten* von auftretenden Fehlern zu machen. Kann beispielsweise ein Verfahren die Fehlertoleranz eines Systems nur sicherstellen, wenn ein Fehler nicht n-mal unmittelbar hintereinander auftritt, so kann dies durch ein Prädikat über einen Orakelstrom oc durch

$$\neg\ \exists\, k \ \bullet\ \ \forall\, j \bullet 0 \leq j \leq (n-1) \ \Rightarrow\ oc.(k+j) = \mathsf{true}$$

ausgedrückt und dann in einem Beweis als Voraussetzung verwendet werden.

5.3 Fehlererkennung

Ein wichtiger Schritt bei der Behandlung von Fehlern ist ihre Erkennung. Wir können unterscheiden zwischen den in Kapitel 2.1.1 drei diskutierten Fehlerbegriffen.

Eine *Fehlerursache* als der eigentliche Defekt eines Systems kann als solcher nicht vom System erkannt werden, da eine Fehlerursache beispielsweise im Entwicklungsprozeß, in fehlerhaften oder unvollständigen Anforderungen oder einer ungeeigneten Architektur ihren Ursprung haben kann. Das System selbst kann diese Defekte nicht direkt erkennen, sondern kann nur deren Wirkung wahrnehmen, die als Versagen oder Fehlerzustand auftreten.

Das *Versagen* eines Systems kann von außen erkannt werden, wenn ein beobachtetes Ist-Verhalten von einem spezifizierten Soll-Verhalten abweicht. Die Erkennung findet dabei also in einem anderen System oder in einer anderen Komponente eines Systems statt. Die Erwartung an die möglicherweise versagende Komponente kann in der erkennenden Komponente als eine Umgebungsannahme formuliert werden. Ein Fehler wird also nicht unbedingt als solcher erkannt, da im allgemeinem bei einer nicht erfüllten Umgebungsannahme das Verhalten nicht spezifiziert ist. Eine Fehlererkennung mit einer zugehörigen Fehlerbehandlung kann in ein System integriert werden, indem die Fehlererkennung mit einer zusätzlichen sinnvollen Reaktion verbunden wird. In Abschnitt 5.6 wird dies diskutiert.

Ein *Fehlerzustand* tritt *innerhalb* eines Systems und *zur Laufzeit* auf. Ein System, das zur Erkennung eines Fehlerzustandes in der Lage ist, muss also die Abweichung des aktuellen Systemzustandes von einem erwünschten Systemzustand selbst erkennen können. Wir wollen nun eine mögliche Modellierung einer Fehlererkennung im Rahmen unseres Systemmodells vorstellen.

Dazu definieren wir ein (Zustands-) Prädikat Ψ (mit $free(\Psi) \subseteq I \cup O \cup A$), das das Vorliegen eines Fehlerzustandes $\alpha \in ERROR(\mathcal{S}, \mathcal{M})$ in einem System $\mathcal{S}$ mit dem Fehlermodell $\mathcal{M}$ anzeigt. Das Prädikat muss die vier Eigenschaften der Auswertbarkeit, Korrektheit, Lebendigkeit und Stabilität aufweisen:

- Das Prädikat Ψ kann *durch das System selbst* ausgewertet werden. Es darf also nicht Variablen enthalten, die der Formalismus zwar zur Verwendung in Beweisen bereitstellt, auf die das System selbst aber nicht zugreifen kann. Dazu gehören beispielsweise die Variablen i^+ (für alle $i \in I$) für noch nicht gelesene Eingaben oder Variable *oc* für Orakel, aber auch bereits gelesene Eingaben i° und sogar vom System bereits gesendete Ausgaben o. Die entsprechenden Informationen stehen dem System nur dann zur Verfügung, wenn sie vom System explizit in lokalen Variablen gespeichert werden.

- Das Prädikat ist *korrekt*. Das Prädikat wird also nur dann wahr, wenn tatsächlich ein Fehler vorliegt. In unserem Formalismus können wir dies formulieren als

$$\alpha \models \Psi \Rightarrow \alpha \in ERROR(\mathcal{S}, \mathcal{M})$$

- Das Prädikat ist *lebendig* in dem Sinne, dass ein Fehler auch tatsächlich erkannt wird, wenn er dauerhaft vorliegt. Wir fordern also nicht, dass ein Fehler sofort erkannt wird, was sich in einer Äquivalenz anstelle der Implikation in der Korrektheitsforderung widerspiegeln würde. Wir fordern nur die abgeschwächte Form

$$\xi.k \in ERROR(\mathcal{S}, \mathcal{M}) \Rightarrow \exists j \geq k \bullet (\xi.j \models \Psi \vee \xi.j \notin ERROR(\mathcal{S}, \mathcal{M}))$$

- Ist ein Fehler durch Ψ erkannt, bleibt Ψ *stabil*, also solange wahr, bis dieser Fehler nicht mehr vorliegt:

$$\xi.k \models \Psi \Rightarrow \xi.(k+1) \models \Psi \vee \xi.(k+1) \notin ERROR(\mathcal{S}, \mathcal{M})$$

Als Beispiel greifen wir auf den Puffer zurück und definieren für ihn ein Fehlermodell zusammen mit einem Fehlererkennungsprädikat.

Beispiel 5.2 Erkennung von Datenverlust im Puffer

In Beispiel 4.12 haben wir eine Spezifikation des Puffers definiert, in der die Daten in einer Variablen q gespeichert werden. Wir wollen nun den potentiellen Datenverlust von Daten aus q modellieren und ein Prädikat Ψ definieren, das diesen Fehler aufdeckt.

Der Datenverlust wird durch die Ergänzung einer Transition τ_3 dargestellt, die das erste Datum in q entfernt:

Name	*Pre*	*Input*	*Output*	*Post*
τ_3	$\#q > 0$	–	–	$q' = rt.q$

Da diese Transition immer schaltbereit ist, wenn Daten in q vorhanden sind, und da jedes Datum vor einer potentiellen Ausgabe einmal ganz vorne in q steht, kann jedes Datum verloren gehen. Durch eine Verstärkung der Vorbedingung könnte man den Verlust auf bestimmte

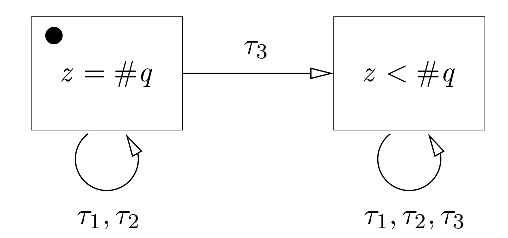


Abbildung 5.3: Invariantendiagramm für *Buffer*

Daten einschränken, oder auch durch Orakelströme Einschränkungen über die Häufigkeit von Datenverlust ausdrücken.

Ohne aufgetretenen Datenverlust sind in q genau $\#i^\circ - \#o$ Nachrichten enthalten, also genau die Differenz zwischen der Zahl der bereits empfangenen Daten und der schon wieder ausgegebenen Daten. Ein Verlust ist also daran erkennbar, dass

$$\#q \neq \#i^\circ - \#o$$

Weder i° noch o sind allerdings Größen, auf die das System zugreifen darf. Sie sind zwar in unserem Formalismus verfügbar, aber eine Implementierung unseres Puffers entsprechend seiner Spezifikation ist nicht in der Lage, sich beispielsweise an alle ausgegebenen Nachrichten in o zu „erinnern“.

Wir entwickeln den Puffer daher weiter, und führen eine neue, mit 0 initialisierte Variable z als Zähler ein. Die beiden Transitionen τ_1 und τ_2 werden verfeinert, indem sie die Variable z nicht mehr beliebig verändern dürfen, sondern in ihr die Länge von q speichern. Die Fehlertransition τ_3 läßt z aber unverändert:

Name	*Pre*	*Input*	*Output*	*Post*
τ_1	$\#q < N$	$i?d$	–	$q' = q \frown \langle d \rangle \wedge z' = z + 1$
τ_2	$\#q > 0$	$i?R$	$o!ft.q$	$q' = rt.q \wedge z' = z - 1$
τ_3	$\#q > 0$	–	–	$q' = rt.q \wedge z' = z$

Das System kann nun auf die Variable z zugreifen, so dass wir folgende Definition von Ψ wählen können:

$$\Psi \overset{df}{=} \#q \neq z$$

Da wir (mit den Beweistechniken aus 3.8.2) zeigen können, dass $z = \#i^\circ - \#o$ eine Invariante unseres fehlerbehafteten Puffers darstellt, ist Ψ eine angemessene Charakterisierung des Datenverlusts, d.h. mit der Variablen z und den Verfeinerungen der Transitionen wurde das System auf geeignete Weise ergänzt, um einen Zustand $\#q \neq \#i^\circ - \#o$ vom System erkennbar zu machen. Die Variable z stellt *Redundanz* entsprechend 2.2.2 dar, die die Fehlererkennung durch das System selbst ermöglicht.

Wir müssen nun noch die Korrektheit, Lebendigkeit und Stabilität von Ψ nachweisen. In unserem Fall gilt eine stärkere Aussage, die diese drei Eigenschaften impliziert, denn der Puffer befindet sich in einem (durch τ_3 verursachten) Fehlerzustand *genau dann, wenn* in diesem Zustand Ψ gilt, also

$$\alpha \in \mathit{Error}(\mathit{Buffer}, (\varnothing, \{\tau_3\})) \quad \Leftrightarrow \quad \alpha \models \Psi \qquad (*)$$

Korrektheit folgt daraus unmittelbar. Lebendigkeit zeigt sich mit der Wahl $j = k$, so dass das erste Disjunktionsglied erfüllt ist. Mit $\xi.(k + 1) = \alpha$ ergibt sich sofort die Disjunktion der Stabilitätsaussage.

Den Beweis von (*) kann man auf das Invariantendiagramm in Abbildung 5.3 abstützen. Daran ist einerseits erkennbar, dass nach der Ausführung der Fehlertransition τ_3 das Prädikat Φ erfüllt ist, und dass dieses andererseits auch *nur* durch Ausführung von τ_3 wahr wird. Es liegt uns hier wiederum ein Beispiel vor, das die Erweiterung von Beweisdiagrammen bei nachträglich modellierten Fehlern illustriert. Für den fehlerfreien Fall ohne die Ausführung der Transition τ_3 zeigt das Diagramm die Aussage $\mathit{Buffer} \Vdash \mathbf{inv}\ z = \#q$. □

In anderen Ansätzen (wie zum Beispiel in [47, 48]) wird Fehlererkennung vereinfacht, indem alle Fehlertransitionen in F eine boolesche Variable f von false auf true setzen müssen. Wurde eine Fehlertransition ausgeführt, ist dies unmittelbar am Wert von f erkennbar. Gegen diesen Ansatz sprechen einige Argumente. Zum einen ist es im allgemeinen praktisch nicht möglich, auftretende Fehlertransitionen sofort durch das System selbst erkennen zu lassen. Beispielsweise ist der Verlust von Daten auf einem Übertragungskanal nicht ohne spezielle Mechanismen erkennbar. Zum anderen können Fehlertransitionen ausgeführt werden, ohne dass dies unbedingt zu einem Fehlerzustand führt. So kann in bestimmten Situationen die Wirkung einer Fehlertransition von einer normalen Transition nicht zu unterscheiden sein, wenn beispielsweise der Wert von Variablen verändert wird, die ohnehin ohne weiteren Lesezugriff neu überschrieben werden.

Dennoch kann es nützlich sein, eine fehleranzeigende Variable f einzuführen, die bei der Verifikation genutzt werden kann, um beispielsweise die Korrektheit eines Fehlererkennungsprädikates zu beweisen. Sie gehört aber nicht zu den Variablen, auf die das System selbst zugreifen kann, darf also nicht Teil von Ψ sein.

Eine pragmatische Realisierung von fehlererkennenden Prädikaten im Rahmen konkreter Programmiersprachen wird in [55] diskutiert. Darin wird gezeigt, wie die Gültigkeit von Invarianten und Vor- und Nachbedingungen von Prozeduraufrufen mit programmiertechnischen Mitteln überwacht wird und wie auf Verletzungen geforderter Eigenschaften mit Hilfe von *Exceptions* reagiert werden kann. Definiert man ein fehlererkennendes Zustandsprädikat Ψ bereits auf der abstrakten Ebene des Transitionssysteme, kann es sowohl formal auf Korrektheit, Stabilität und Lebendigkeit untersucht werden als auch mit den programmiersprachlichen Techniken konkret in einer Implementierung verwendet werden.

5.4 Einführung von Fehlermeldungen

Eine einfache, mögliche Reaktion auf einen erkannten Fehler ist die Meldung des erkannten Fehlers nach aussen an die Systemumgebung. Dazu kann ein bestehender oder ein neuer, dedizierter Kanal f verwendet werden. Unter Verwendung der in Kapitel 4.1.1 vorgestellten Technik können wir zunächst einen Kanal ergänzen oder den Typ eines bestehenden Kanals verhaltensneutral erweitern. Ein Fehler werde entsprechend Abschnitt 5.3 durch ein Zustandsprädikat Ψ erkannt. Das Senden einer Fehlermeldung

fail auf dem Kanal f kann dann durch eine neue Transition der Form

Name	*Pre*	*Input*	*Output*	*Post*
τ_f	Ψ	–	$f!fail$	*post*

modelliert werden. Die Nachbedingung *post* fordert typischerweise, dass der lokale Datenzustand bei Versenden der Fehlermeldung unverändert bleibt. Allerdings kann hier auch eine Fehlerkorrektur initiiert werden, wie wir im folgenden Beispiel und im nächsten Abschnitt sehen werden.

Diese neue Transition ändert das Verhalten des Systems nicht, solange kein Fehler auftritt. Damit die Fehlermeldung nur genau dann geschickt wird, wenn ein Fehler vorliegt, dürfen bereits existierende Transitionen die Nachricht *fail* nicht versenden. Wir stellen dies sicher durch das Entfernen aller Transitionen, die dies tun. Dabei sollen alle anderen Nachrichten auf f noch immer gesendet werden können:

$$E = \{(\alpha, \beta) \mid \beta.f \;=\; \alpha.f \frown \langle fail \rangle\}$$

Die Transition τ_f muss bei dieser Modellierung aber nicht sofort ausgeführt werden, wenn ein Fehler vorliegt. Sind noch andere Transitionen schaltbereit, können auch diese gewählt werden. Nur bei permanenten Fehlern wird τ_f aufgrund der Fairness im Ausführungsmodell irgendwann einmal ausgeführt. Bei transienten Fehlern (also nur vorübergehender Gültigkeit des fehlererkennenden Zustandsprädikates Ψ) kann es passieren, dass eine Fehlernachricht nie gesendet wird. Ferner ist es möglich, dass eine Fehlermeldung mehrfach ausgegeben wird. Wir wollen nun zwei verfeinerte Modellierungen für sofortiges Senden und für einmaliges Senden einer Fehlermeldung angeben.

Unmittelbare Fehlermeldung Eine Fehlermeldung wird sofort ausgegeben, wenn wir die Schaltbereitschaft anderer Transitionen bei erkanntem Fehler aufheben. Dies geschieht durch Entfernen von

$$E_1 = \{(\alpha, \beta) \mid \alpha \models \Psi\}$$

Damit beschreibt $\mathcal{M} = (\{\tau_f\}, E \cup E_1)$ die Modifikation, die ein System bei erkanntem Fehler sofort eine Fehlermeldung ausgeben läßt.

Einmalige Fehlermeldung Soll eine Fehlermeldung nur einmal gesendet werden, so darf τ_f nur dann schaltbereit sein, wenn der Fehler noch nicht gemeldet wurde. Dazu wählen wir eine neue boolesche Variable *sent*, die mit false initialisiert wird und vermöge

$$E_2 = \{(\alpha, \beta) \mid \alpha.sent \neq \beta.sent\} \setminus \{(\alpha, \beta) \mid \alpha.\Psi \wedge \neg\, \beta.\Psi \wedge \neg\, \beta.sent\}$$

von keiner anderen Transition verändert wird. Nur wenn ein Fehlerzustand nicht mehr vorliegt, was am Übergang von Ψ zu false erkennbar ist, soll der Wert von *sent* wieder auf den Initialwert gesetzt werden. Mit τ'_f als

Name	*Pre*	*Input*	*Output*	*Post*
τ'_f	$\Psi \wedge \neg\, sent$	–	$f!fail$	$sent' = \mathsf{true}$

beschreibt die Modifikation $\mathcal{M} = (\{\tau_f'\}, E \cup E_2)$ ein System, in dem eine Fehlermeldung nur einmal gesendet werden kann.

Beide Modifikationen lassen sich kombinieren. Ein durch $\mathcal{M} = (\{\tau_f'\}, E \cup E_1 \cup E_2)$ modifiziertes System gibt eine Fehlermeldung sofort aus, bleibt dann aber in einem Fangzustand stehen, da es keine Transitionen mehr gibt, die bei gültigem Ψ schaltbereit sind. Durch Hinzufügen von weiteren, korrigierend eingreifenden Transitionen kann ein System wieder aus dem Fehlerzustand herausgeführt werden. Wir diskutieren dies im nächsten Abschnitt.

Beispiel 5.3 Fehlermeldung bei Datenverlust des Puffers

Der Verlust von Daten im Puffer wird in Beispiel 5.2 durch das Prädikat $\Psi \equiv \#q \neq z$ erkannt. Um die Fehlermeldung auf dem Ausgabekanal o ausgeben zu können, erweitern wir seinen Typ um das Element *fail*, also $type(o) = D \cup \{fail\}$. Wir ergänzen standardmäßig die Transition

Name	*Pre*	*Input*	*Output*	*Post*
τ_f	$\#q \neq z$	–	$o!fail$	*post*

Damit diese Fehlermeldung nur einmal pro verlorenem Datum gesendet wird, modifizieren wir τ_f derart, dass wieder ein konsistenter Zustand hergestellt wird, also Ψ nicht mehr gilt. Dazu wählen wir die Nachbedingung *post* so, dass q unverändert bleibt, und z mittels $z' = \#q$ angepasst wird. Im Einzelfall kann also auf die Verwendung einer neuen Variable *sent* verzichtet werden und dennoch sichergestellt sein, dass eine Fehlermeldung nur einmal geschickt wird.

Damit die Fehlermeldung unmittelbar erfolgt, verstärken wir auf schematische Weise entsprechend E_1 die Vorbedingungen der Transitionen. Insgesamt wird der Puffer, der die Fehler erkennt und darauf sofort mit einer Fehlermeldung sowie einer einfachen Fehlerkorrektur reagiert, folgendermaßen spezifiziert:

Name	*Pre*	*Input*	*Output*	*Post*
τ_1	$\#q < N \wedge \#q = z$	$i?d$	–	$q' = q \frown \langle d \rangle \wedge z' = z + 1$
τ_2	$\#q > 0 \wedge \#q = z$	$i?R$	$o!ft.q$	$q' = rt.q \wedge z' = z - 1$
τ_3	$\#q > 0 \wedge \#q = z$	–	–	$q' = rt.q \wedge z' = z$
τ_f	$\#q \neq z$	–	$o!fail$	$q' = q \wedge z' = \#q$

In einer Realisierung des Puffers wird die Fehlertransition τ_3 natürlich nicht realisiert; sie repräsentiert nur das ungewollte, jedoch als unvermeidbar eingeschätzte Fehlerverhalten. □

5.5 Fehlerkorrektur

Die frühzeitige Berücksichtigung und Modellierung von Fehlern bereits während der Entwicklungszeit ermöglicht es, Maßnahmen und Techniken zur *Fehlerkorrektur* in ein System zu integrieren, die während der Laufzeit des Systems die Auswirkungen der Fehler verbergen oder zumindest eindämmen. Wir geben hier zwei Techniken an, die sich zur Fehlerkorrektur im Rahmen unseres Systemmodells eignen: Die Ergänzung einer korrigierenden Transitionsmenge und die Verwendung von korrigierenden Treiberkomponenten.

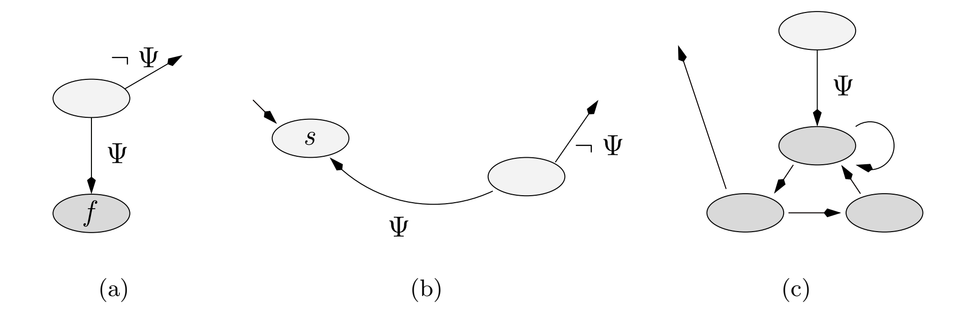


Abbildung 5.4: Varianten korrigierender Transitionen

5.5.1 Ergänzung von korrigierenden Transitionen

Ist ein Fehler erkannt, so soll auf diesen angemessen und sinnvoll reagiert werden. Die Ausgabe einer Fehlermeldung, wie wir sie im vorigen Abschnitt diskutiert haben, stellt bereits ein Beispiel einer einfachen Reaktion dar. Wir wollen hier nun allgemeiner beschreiben, wie Systeme um Reaktionen ergänzt werden können.

Es ist typischerweise nicht ausreichend, lediglich neue Transitionen hinzuzufügen, um eine gewünschte Reaktion auf Fehler ausdrücken zu können. Es kann nötig sein, neue lokale Variablen zu ergänzen oder die Wertemenge (den Typ) von vorhandenen Variablen zu erweitern. Insbesondere ist es damit möglich, neue Kontrollzustände aufzunehmen. Mit den Techniken aus Abschnitt 4.1 sind wir in der Lage, die benötigten neuen Variablen oder erweiterten Typen verhaltensneutral zu ergänzen. Wir gehen daher im folgenden davon aus, dass alle Variablen mit entsprechenden Wertebereichen bereits vorhanden sind.

Wir spezifizieren exemplarisch drei Varianten von Reaktionen auf einen entdeckten Fehler, die in Abbildung 5.4 dargestellt sind: Der Sprung in einen Fangzustand, der Neustart des Systems und der Start einer frei wählbaren Fehlerbehandlung, die eine Verallgemeinerung der ersten beiden Reaktionen darstellt.

Sprung in einen Fangzustand In Abbildung 5.4(a) ist der Sprung in einen Fangzustand f illustriert: Wird ein Fehler durch ein Prädikat Ψ erkannt, dürfen Normaltransitionen nicht mehr gewählt werden, da ihre Vorbedingung um $\neg\,\Psi$ verstärkt wird. Stattdessen springt das System in den Fangzustand, von dem aus keine Transitionen mehr möglich sind, so dass insbesondere keine Ein- und Ausgaben mehr stattfinden. Damit bleiben automatisch alle Sicherheitseigenschaften des Systems erhalten.

Der Sprung in den Fangzustand muss von allen erreichbaren Zuständen des Systems möglich sein. In einer graphischen Repräsentation eines Systems stellt sich wie in Abbildung 5.4(a) der Fangzustand als ein neuer Knoten f dar. Zu diesem führen von allen anderen Knoten des Systems jeweils eine Transition, während von ihm keine Transition mehr ausgeht. Für kleinere Systeme ist diese Darstellung sehr intuitiv. Für große, komplexe Systeme mit zahlreichen Kontrollzuständen ist diese Darstellung allerdings nicht

gut geeignet, da viele neue Kanten ergänzt werden müssen. Wir können den Sprung in einen Fangzustand auch sehr kompakt als Modifikation $\mathcal{M}$ ausdrücken.

Wir benötigen dazu eine neue boolesche Variable *trap*, die initial den Wert false besitzt, für die also gelten soll

$$\Upsilon \Rightarrow \neg\, trap$$

Wird ein Fehler durch ein Prädikat Ψ erkannt, soll – vollkommen unabhängig vom Zustand des Systems – in den Fangzustand gesprungen werden. Dies wird erreicht durch eine Modifikation mit

$$F \;=\; \begin{array}{ccccc} Name & Pre & Input & Output & Post \\ \hline \tau_{trap} & \Psi & - & - & trap \end{array}$$

Der Sprung in den Fangzustand soll sofort erfolgen, wenn der Fehler entdeckt wurde. Daher sind jene Transitionen zu entfernen, die es erlauben würden, dass das System weiterarbeitet, obwohl der Fehler bereits entdeckt wurde. Da auch im Fangzustand das Prädikat Ψ wahr bleibt, ist sichergestellt, dass keine Transitionen von ihm ausgehen. Es ist also ausreichend, die folgenden Transitionen zu entfernen:

$$E \;=\; \{(\alpha, \beta) \mid \alpha \models \Psi\}$$

Damit beschreibt der Ausdruck $\mathcal{S}\triangle(E, F)$ für jedes System $\mathcal{S}$ mit dem Fehlerprädikat Ψ das System, das bei Auftreten eines an Ψ erkennbaren Fehlers seine Aktivität einstellt. Die sogenannten *fail-safe* Systeme lassen sich mit dieser Technik modellieren.

Im Unterschied zu Fehlermodellen wie *crash failure* und *fail-stop failure*, die den Halt des Systems bereits mit enthalten, wird hier bewußt der Sprung in den Fangzustand als Reaktion auf einen beliebigen, durch Ψ charakterisierbaren Fehler gewählt.

Neustart des Systems In Abbildung 5.4(b) wird der Neustart eines Systems bei einem auftretenden Fehler dargestellt. Wird ein Fehler durch Ψ erkannt, wird in einen Startzustand s gesprungen.

Soll diese Fehlerbehandlung graphisch repräsentiert werden, gelten die gleichen Vor- und Nachteile wie beim Sprung in den Fangzustand: Während die graphische Darstellung bei kleineren Systemen sehr intuitiv ist, wird sie bei komplexen Systemen schnell unübersichtlich. Da es mehrere Knoten geben kann, die Startzustände darstellen, ist es hier sogar nötig, von jedem Knoten des Systems eine Transition zu jedem Startknoten zu ergänzen.

Ein Neustart ist durch eine Modifikation erneut sehr einfach modellierbar. Während F die Transitionen enthält, die den Rücksprung beschreiben, wird durch E das System daran gehindert, bei einem entdeckten Fehler unbeeinflußt seine normale Aktivität fortzusetzen:

$$\begin{aligned} F &= \{(\alpha, \beta) \mid \alpha \models \Psi \wedge \beta \models \Upsilon\} \\ E &= \{(\alpha, \beta) \mid \alpha \models \Psi\} \end{aligned}$$

Das System $\mathcal{S}\triangle\mathcal{M}$ mit $\mathcal{M} = (E, F)$ spezifiziert damit ein System, das sich von $\mathcal{S}$ darin unterscheidet, dass es bei entdecktem (durch Ψ beschriebenem) Fehler neu startet.

Ergänzen einer Ausnahmebehandlung Abbildung 5.4(c) illustriert den allgemeinsten Fall einer Fehlerbehandlung: Nach der Entdeckung eines Fehlers wird eine Fehlerbehandlung gestartet, die als eigenes Transitionssystem spezifiziert ist. Ist die Fehlerbehandlung abgeschlossen, erfolgt ein Rücksprung in das ursprüngliche System. Diese Reaktion entspricht einer *Ausnahmebehandlung*, wie man sie aus Programmiersprachen als *exceptions* kennt.

Wir beschreiben eine Ausnahmebehandlung mit Hilfe einer Transitionsmenge C und zwei Zustandsprädikaten *Entry* und *Exit*. Das Prädikat *Entry* beschreibt die zu erfüllende Startbedingung der Ausnahmebehandlung und umfaßt eine Auszeichnung des Kontrollzustandes, in dem das System gestartet wird wie auch die Initialisierung von lokalen Variablen. *Exit* charakterisiert die Zustände, von denen aus der Rücksprung in das Normalsystem erfolgen darf. Wir zeichnen eine bislang noch nicht im System verwendete Variable xp aus, mit der es möglich ist, die Normaltransitionen des Systems während der Ausnahmebehandlung zu deaktivieren.

Eine Ausnahmebehandlung wird mit einem Transitionssystem verknüpft, indem zwei Prädikate Ψ und Φ definiert werden. Dabei beschreibt Ψ einen Fehler- oder Ausnahmezustand, in dem die Ausnahmebehandlung gestartet werden soll, während Φ die Zustände des Systems charakterisiert, in die nach der Ausführung der Ausnahmebehandlung zurückgesprungen wird. Formal können wir dies folgendermaßen definieren:

Definition 5.1 *Ausnahmebehandlung in Transitionssystemen*

Seien ein Transitionssystem $\mathcal{S} = (I, O, A, \Upsilon, T)$ *mit* $xp \notin A$ *und* $type(xp) = \{\mathsf{true}, \mathsf{false}\}$ *sowie zwei Prädikate* Ψ *und* Φ *gegeben.*

Eine Ausnahmebehandlung *wird durch eine Transitionsmenge C und die zwei Zustandsprädikate Entry und Exit definiert. Dabei gelte*

$$\forall (\alpha, \beta) \in C \bullet \alpha \models xp \wedge \beta \models xp$$

Die Erweiterung von $\mathcal{S}$ um die Ausnahmebehandlung, die bei Gültigkeit von Ψ gestartet wird und die nach Ausführung in einen Zustand zurückspringt, für den Φ gilt, ist definiert durch

$$\mathcal{S} \triangle (E, F)$$

mit

$$\begin{aligned}
F = \;& C \;\cup \\
& \{\, (\alpha,\beta) \mid \;\beta \models xp \wedge \alpha \models \Psi \wedge \beta \models Entry \\
& \qquad\qquad \wedge \forall \beta' \bullet Agree(\alpha,\beta') \subseteq Agree(\alpha,\beta) \,\} \;\cup \\
& \{\, (\alpha,\beta) \mid \;\beta \models \neg\, xp \wedge \alpha \models Exit \wedge \beta \models \Phi \\
& \qquad\qquad \wedge \forall \beta' \bullet Agree(\alpha,\beta') \subseteq Agree(\alpha,\beta) \,\} \\
E = \;& \{\, (\alpha,\beta) \mid \;\alpha \models xp \,\} \;\cup\; \{\, (\alpha,\beta) \mid \alpha \models \Phi \,\}
\end{aligned}$$

Dabei sei xp in diesem System mit false *initialisiert.* □

Diese Definition bedarf einer Erläuterung. Die Variable xp wird beim Eintritt in die Ausnahmebehandlung auf true gesetzt, beim Verlassen wieder auf false, ansonsten ändert sie ihren Wert nicht. Die Transitionen des Systems $\mathcal{S}$ sind (wegen E) während der aktiven Ausnahmebehandlung also nicht schaltbereit, und nur die Transitionen aus C können schalten. Das Prädikat Ψ eignet sich nur bedingt, um Transitionen von T während einer Ausnahmebehandlung zu vermeiden. Das Ziel der Abarbeitung wird es oft sein, einen Fehlerzustand zu korrigieren und damit Ψ ungültig zu machen. Transitionen aus T könnten also zu früh wieder schaltbereit werden. Daher ist eine neue Variable xp notwendig.

Beim Start der Ausnahmebehandlung wird eine Transition ausgeführt, die in einen Zustand β führt, für den das Prädikat *Entry* gilt, der aber ansonsten dem Ausgangszustand α *möglichst ähnlich* ist. Ähnlichkeit wird hierbei verstanden als eine maximale Übereinstimmung in der Belegung aller Variablen. Typischerweise wird das Prädikat *Entry* den Kontrollzustand σ auf einen neuen Wert setzen, und gegebenenfalls Variablen initialisieren, aber sonst keine Aussagen über die Variablen des Systems machen. Mit obiger Definition bleiben diese Variablen bei Ausführung dieser Transition tatsächlich unverändert, wie man es auch bei einer operationellen Sicht erwarten würde: Die expliziten Zuweisungen werden erfüllt, während alle übrigen Variablen unangetastet bleiben. Beim Rücksprung aus einem Zustand, in dem *Exit* gilt, in einen Zustand, in dem Φ gilt, bleiben in gleicher Weise möglichst viele Variablenwerte unverändert.

Die Ergänzung einer beliebig wählbaren Ausnahmebehandlung ist die allgemeinste Form des Umgangs mit Fehlern. Damit sind alle denkbaren Reaktionen modellierbar, unter anderem auch Techniken des *Backward-* und *Forward-Recovery* (beschrieben beispielsweise in [44]), mit denen versucht wird, trotz auftretender Fehler möglichst wenig vom Normalablauf eines Systems abzuweichen. Als einfachere Beispiele definieren wir den Sprung in einen Fangzustand und den Neustart eines Systems im formalen Rahmen der Ausnahmebehandlungen:

Beispiel 5.4 Wir können den Sprung in einen Fangzustand folgendermaßen als Ausnahmebehandlung definieren:

$$\begin{aligned} C &\stackrel{df}{=} \varnothing \\ \mathit{Entry} &\stackrel{df}{=} \mathsf{true} \\ \mathit{Exit} &\stackrel{df}{=} \mathsf{false} \end{aligned}$$

Durch die Variable xp ist sichergestellt, dass keine Transitionen des Systems schaltbereit sind. Sie übernimmt die Rolle von *trap*. Da es keinen Rücksprung gibt, ist Φ beliebig definierbar.

Auch der Neustart des Systems kann leicht definiert werden. Wir benötigen in C erneut keine Transitionen, da wir die Ausnahmebehandlung sofort wieder verlassen wollen. Das System springt mittels Υ als Φ in einen Startzustand von $\mathcal{S}$ zurück. Wir haben also

$$\begin{aligned} C &\stackrel{df}{=} \varnothing \\ \mathit{Entry} &\stackrel{df}{=} \mathsf{true} \\ \mathit{Exit} &\stackrel{df}{=} \mathsf{true} \\ \Phi &\stackrel{df}{=} \Upsilon \end{aligned}$$

□

In allen vorgestellten Formalisierungen wird bei einem erkannten Fehler *sofort* die Fehlerbehandlung gestartet. Es kann jedoch in konkreten Anwendungsfällen sinnvoller sein, die normale Ausführung eines Systems noch etwas weiterzuführen, da gewisse Teilaufgaben noch vollendet werden sollen und müssen, bevor die Fehlerbehandlung gestartet wird. Dies ist durch eine geeignete Anpassung des Prädikates Ψ modellierbar.

Die Fehlerbehandlungen haben die Eigenschaft, dass sie keinen Einfluß auf das Systemverhalten haben, solange keine Fehler auftreten. Dies ist offensichtlich, da Modifikationen nur dann eine Wirkung zeigen können, wenn Φ gilt oder galt. Für eine durch $\mathcal{M}$ beschriebene Ausnahmebehandlung und ein System $\mathcal{S}$, in dem $\neg\,\Psi$ eine Invariante ist, gilt also

$$\langle\!\langle \mathcal{S} \rangle\!\rangle = \langle\!\langle \mathcal{S} \triangle \mathcal{M} \rangle\!\rangle$$

Selbstverständlich muss diese Eigenschaft nicht immer gelten. So kann beispielsweise ein System erweitert werden um Transitionen, die regelmäßig relevante Teile des Systemzustandes abspeichern, um im Fehlerfalle auf einen korrekten Zustand zurückgreifen zu können und ihn wiederherzustellen. Die Strategie des *Backward Recovery* verwendet diese Technik. Die Bereitstellung der notwendigen Redundanz hat Einfluß auf das Normalverhalten des Systems und macht es typischerweise ein wenig langsamer. Auch diese Techniken können durch die Ergänzung von Transitionen modelliert werden. Wir haben uns hier auf die Darstellung einiger einfacher Beispiele beschränkt.

5.5.2 Treiberkomponenten

Es ist wünschenswert, die Auswirkungen von Fehlern nach Möglichkeit lokal zu halten, um eine Ausbreitung über weitere Teile eines Systems zu vermeiden. Dies kann durch sogenannte *Treiberkomponenten* geschehen, die zwischen einer durch Fehler beeinträchtigten Komponente und ihrer Umgebung geschaltet werden. Je nach Art und Schwere des Fehlers sind verschiedene Arten von Treibern notwendig. Die Treiber haben dieselbe Gestalt wie die Modifikationskomponenten, und werden daher auch wie in Abbildung 4.3 dargestellt. Je nach Typ des Fehlers kann es genügen, dass Treiber ausschließlich die Ausgaben kontrollieren und korrigieren, wobei sie unter Umständen Kenntnis über die Eingaben besitzen müssen. Im allgemeinsten Fall muss ein Treiber sowohl auf Ein- als auch auf Ausgabekanäle Einfluß nehmen. Alle Varianten aus Abbildung 4.3 sind also auch für Treiber relevant.

Während die Modifikationskomponenten in Kapitel 4 zur Modellierung von Fehlern verwendet wurden, können sie ebenso der Darstellung von Treibern dienen. Der wesentliche Unterschied besteht darin, dass die Modifikationskomponenten nicht implementiert werden, sondern nur verwendet werden, um die Abweichungen des Verhaltens zu modellieren. Treiberkomponenten werden dagegen tatsächlich realisiert. Werden, wie in Abbildung 5.5 dargestellt, Fehler einer Komponente S durch eine Modifikationskomponente $\mathcal{M}$ modelliert, die durch eine Treiberkomponente C korrigiert werden können, so heben sich die Wirkungen von C und $\mathcal{M}$ gegenseitig auf und es gilt idealerweise (nach geeigneter Umbenennung von Kanälen)

$$S \rightsquigarrow C \otimes \mathcal{M} \otimes S$$

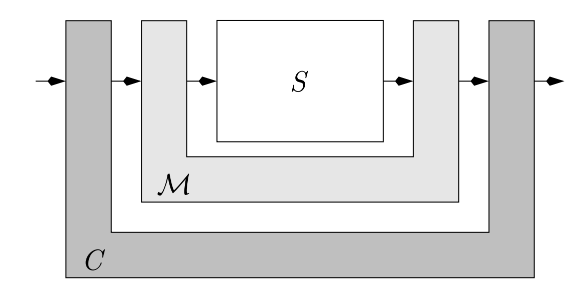


Abbildung 5.5: Kombination von Modifikations- und Treiberkomponente

Beispiel 5.5 Ein klassisches Beispiel für den sinnvollen Einsatz von Treibern stellen verlustbehaftete Kommunikationskanäle dar. Ein idealisierter Übertragungskanal *Channel* sei absolut fehler- und verlustfrei. Er sei parametrisierbar, reagiere also auf gewisse Steuersignale mit unterschiedlichen Qualitätsmerkmalen (wie Datenrate oder Verzögerungszeiten).

Eine realistische Implementierung des Kanals kann jedoch dem Verlust von Nachrichten ausgesetzt sein. Dies können wir modellieren durch Hinzufügen einer (virtuellen) Komponente *Loss*, die den sporadischen Verlust von Nachrichten spezifiziert. Es genügt, diese Komponente auf einer Seite des Kanals zu ergänzen, da es beobachtungsäquivalent ist, ob eine Nachricht sofort verloren wird oder erst *Channel* passiert und dann verloren geht.

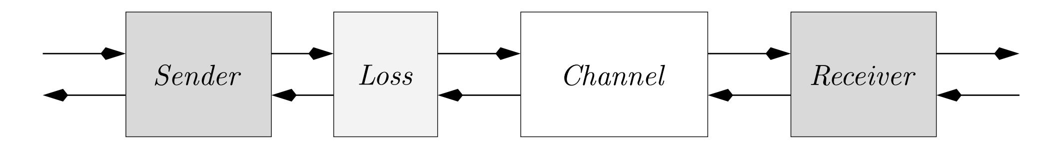


Abbildung 5.6: Fehlerhafter Übertragungskanal mit Treibern

Zum Ausgleich dieses Fehlers ergänzen wir entsprechend Abbildung 5.6 das System um die zwei Treiberkomponenten *Sender* und *Receiver*, die mit Hilfe eines Protokolls sicherstellen, dass trotz auftretender Nachrichtenverluste alle Nachrichten durch entsprechende Wiederholungen übertragen werden. Es gibt eine Vielzahl von Protokollen (beschrieben beispielsweise in [67]), die verschiedene Arten von Fehlern wie die Generierung von Nachrichten oder die Vertauschung ihrer Reihenfolge tolerieren können. Mit Hilfe von Orakelströmen aus Abschnitt 5.2 sind wir in der Lage auszudrücken, dass nicht alle Nachrichten verloren gehen – dies würde die Übertragung schließlich unterbrechen. Kann man in unserem Fall

$$\mathit{Channel} \rightsquigarrow \mathit{Sender} \otimes \mathit{Loss} \otimes \mathit{Channel} \otimes \mathit{Receiver}$$

nachweisen, so ist ein gewähltes Protokoll für ein vorausgesetztes Fehlermodell angemessen. □

Im Extremfall kann der Treiber die volle, geforderte Funktionalität von S übernehmen und die fehlerhaften Ausgaben von $\mathcal{M} \otimes S$ dabei ignorieren. Ein Beispiel stellen die in Kapitel 2.2.2 beschriebenen Reservekomponenten dar, die in unserem Systemmodell formal mit dem Konzept der Treiberkomponenten eingeführt werden können.

5.6 Erhöhung der Systemrobustheit

In Kapitel 4.6 wurden Fehler diskutiert, die nicht im System selbst, sondern in der Umgebung eines Systems auftreten. Das betroffene System erhält dann Eingaben, von denen angenommen wurde, dass sie nie empfangen werden. Das Verhalten des Systems kann in einem solchen Fall nicht spezifiziert sein, so dass es sich auf unvorhersehbare Weise verhalten kann oder beispielsweise seine Aktivität schlicht beendet.

Im Laufe eines Entwicklungsprozesses, in dem Fehler (unter Umständen erst nachträglich) modelliert werden, muss es möglich sein, ein System in einer Art und Weise zu modifizieren, dass es auch in einer fehlerhaften Umgebung noch weitgehend sinnvoll, d.h. auf definierbare Weise reagiert. Wir wollen also die *Robustheit* eines Systems erhöhen.

Wir werden im folgenden diskutieren, wie Entwicklungsschritte zur Erhöhung der Robustheit formal in unserem Systemmodell dargestellt werden können. Dazu unterscheiden wir wie in Kapitel 4.6 nach expliziten und impliziten Umgebungsannahmen.

5.6.1 Explizite Umgebungsannahmen

Sind die Annahmen an eine Systemumgebung explizit formuliert, so können sie durch ein Prädikat A (mit freien Variablen aus I) dargestellt werden. Das Verhalten der Komponente selbst wird im Rahmen von A/G-Spezifikationen dann durch ein Prädikat G repräsentiert.

Fehler in der Umgebung führen in diesem Kontext zu Eingabeströmen, die die Annahme A nicht erfüllen. Typischerweise interessiert man sich nicht für alle denkbaren Fehler, die sich durch eine Negation von A charakterisieren ließen, sondern nur für eine gewisse Teilmenge dieser, für die die Systemrobustheit erhöht werden soll. Wir können diese gewählte Teilmenge der externen Fehler mit Hilfe einer *Abschwächung* A' der Umgebungsannahme formulieren, für die also $A \Rightarrow A'$ gilt.

Das Verhalten des Systems wollen wir nun verändern zu G', so dass es auch bei Eingaben, die zwar A', aber nicht mehr A erfüllen, ein definiertes Verhalten zeigt. Welches konkrete Verhalten ein Entwickler hier wählt, ist unmittelbar von der Anwendung abhängig. Das neue Verhalten G' muss aber die Forderung erfüllen, dass es sich nicht vom ursprünglichen Verhalten unterscheidet, wenn die Eingaben die Annahme A erfüllen. Gilt also A und G', so muss auch G erfüllt sein.

Während das Verhalten bei auftretenden externen Fehlern im ursprünglichen System (A/G) unspezifiziert und damit beliebig ist, reagiert das System (A'/G') auf wohldefinierte Weise. Damit ist im System (A'/G') der Nichtdeterminismus reduziert, und dieser Entwicklungsschritt stellt eine Verhaltensverfeinerung im Sinne von Abschnitt 3.9.1 dar. Folgende (einfach beweisbare) Regel faßt diese Aussage zusammen:

$$\begin{array}{|l} A \Rightarrow A' \\ A \Rightarrow (G' \Rightarrow G) \\ \hline (A, G) \leadsto (A', G') \end{array}$$

Bei Black-Box Spezifikationen mit expliziten Umgebungsannahmen läßt sich die Systemrobustheit also einfach durch eine Verstärkung des verhaltensbeschreibenden Prädikates darstellen, das zusätzlich eine Aussage über das Systemverhalten im Fehlerfalle macht.

Beispiel 5.6 Unterlauf des Puffers (A/G-Spezifikation)

Wir erhöhen die Robustheit des Puffers, der in Beispiel 3.4 im Format einer A/G-Spezifikation definiert wurde. Dazu wollen wir Eingabeströme zulassen, in denen Anfragen an den Puffer geschickt werden, ohne dass vorher ausreichend viele Daten gesendet wurden. In einem solchen Fall tritt also ein sogenannter *Unterlauf* des Puffers auf. Damit das Verhalten des Puffers in einem derartigen Fall nicht mehr unspezifiziert ist, definieren wir eine naheliegende Reaktion: Im Falle eines Unterlaufs soll die entsprechende Anfrage ignoriert werden. Diese Änderung drücken wir nun formal aus.

Da wir die Annahme fallen lassen, dass in jedem Präfix die Zahl der Anfragen nicht größer als die Zahl der Daten ist, schwächen wir die Annahme A ab und fordern von der Umgebung nur noch

$$A' \stackrel{df}{=} \quad \forall\, i' \sqsubseteq i \bullet \#D Ⓢ i' - \#\{R\} Ⓢ i' \leq N$$

Wir passen die Spezifikation des Systems an zu

$$G' \stackrel{df}{=} \quad o \sqsubseteq D Ⓢ i \;\wedge\; \#o = \#\{R\} Ⓢ i - underflow(i)$$

Die ausgegebenen Daten sind noch immer genau jene Daten, die auch empfangen werden; allerdings wird nicht mehr für jede Anfrage eine Ausgabe gemacht: Die Länge der Ausgabe wird um die Anzahl der auftretenden Unterläufe reduziert. Die Funktion *underflow* wird definiert durch Abstützung auf die Hilfsfunktion *uf* mit

$$underflow(i) \quad = \quad uf(i, 0)$$

Die Funktion *uf* ermittelt für einen Eingabestrom die Zahl der Unterläufe, die er auslösen wird. Dazu zählt sie im zweiten Argument die im Puffer gespeicherten Daten. Diese Zahl erhöht sich demzufolge beim Empfang eines Datums und reduziert sich bei Ausgabe eines Datums. Sind keine Daten mehr vorhanden, wird aber eine Anfrage empfangen, liegt ein Unterlauf vor. Die Funktion *uf* können wir also rekursiv definieren durch

$$\begin{array}{lcl} uf(\langle\rangle, n) & = & 0 \\ uf(\langle d\rangle \frown i, n) & = & uf(i, n+1) \\ uf(\langle R\rangle \frown i, n) & = & uf(i, n-1) \text{ für } n > 0 \\ uf(\langle R\rangle \frown i, 0) & = & 1 + uf(i, 0) \end{array}$$

Gilt für einen Eingabestrom i die Bedingung

$$\forall\, i' \sqsubseteq i \bullet \#\{R\} Ⓢ i' \leq \#D Ⓢ i'$$

so tritt kein Unterlauf auf, da stets ausreichend viele Daten vorhanden sind. Es gilt dann also $underflow(i) = 0$. Damit ist die ursprüngliche Spezifikation erfüllt, wenn A gilt. Wir haben also

$$A \Rightarrow (G' \Rightarrow G)$$

und somit eine Verfeinerung des Systems spezifiziert, die eine erhöhte Robustheit aufweist. □

5.6.2 Implizite Umgebungsannahmen

Ein Charakteristikum reaktiver Systeme ist es, jederzeit auf Eingaben zu reagieren. In Abschnitt 4.6.2 leiten wir – basierend auf dieser Annahme – aus einem Transitionssystem eine implizite Anforderung an die Umgebung ab: Die (Präfixe von) Eingaben dürfen nicht in Zustände führen, von denen aus die Eingaben nicht mehr vollständig gelesen werden können.

Das Erhöhen der Robustheit eines Transitionssystems besteht folglich aus dem Ergänzen von Transitionen, die die Zahl der Fangzustände reduzieren und damit die Reaktivität des Systems verbessern.

Für ein System $\mathcal{S} = (I, O, A, \Upsilon, T)$ definieren wir dazu eine Menge von Transitionen F, so dass jede Transition $(\alpha, \beta) \in F$ *von einem Fangzustand von* $\mathcal{S}$ *ausgeht*: Es gibt also in T keine Transition, die im Zustand α schaltbereit ist, formal (mit $V = I \cup O \cup A$) beschrieben durch

$$\forall\, i \bullet \alpha.i^\circ \neq \langle\rangle \quad \wedge \quad \forall\, \widehat{\alpha}, \gamma \bullet \alpha \stackrel{V}{=} \widehat{\alpha} \Rightarrow (\widehat{\alpha}, \gamma) \notin T$$

Einen Zustand α nennen wir nicht Fangzustand, wenn nur aufgrund mangelnder Nachrichten auf den Eingabekanälen keine Transition schaltbereit ist. Da die Umgebungstransitionen T^ϵ immer schaltbereit sind, werden im weiteren Systemablauf noch weitere Eingaben produziert. Liegen aber Eingaben an allen Kanälen an und ist keine Transition aus T schaltbereit, so ändert sich dies auch nicht mehr durch ausgeführte Umgebungstransitionen aus T^ϵ.

Sinnvollerweise sollten die neuen Transitionen von erreichbaren Zuständen ausgehen. Es sollte also einen Ablauf geben, in dem der Zustand α auftritt:

$$\exists\, \xi \in \langle\!\langle \mathcal{S} \rangle\!\rangle \bullet \exists\, k \bullet \xi.k \stackrel{V}{=} \alpha$$

Ist ein Zustand nicht erreichbar, kann er auch kein Fangzustand sein. Das Ergänzen einer Transition an einen unerreichbaren Zustand ändert das Systemverhalten und damit auch die Robustheit nicht.

Modifizieren wir ein System $\mathcal{S}$ zu $\mathcal{S}\triangle(\varnothing, F)$, so wird damit seine Robustheit erhöht, da das System weniger Fangzustände aufweist. Natürlich können auch in einem robusteren System noch immer Fangzustände existieren; es können sogar neue Fangzustände hinzukommen, die zuvor unerreichbar waren, aber durch F erreichbar werden. Diese werden jedoch in einem Ablauf erst später erreicht, so dass dennoch sinnvollerweise von einem robusteren System gesprochen werden kann. In welche Zustände die mit F neu ergänzten Transitionen führen, bleibt dem Entwickler überlassen und ist von der konkreten Anwendung abhängig. So können beispielsweise unerwartete Nachrichten, die sonst zu einem blockiertem System geführt hätten, einfach verworfen werden oder es kann eine andere Fehlerbehandlung ergänzt werden, wie wir es schon im Abschnitt 5.5 vorgestellt haben.

Im allgemeinen wird man ein System nicht gegenüber *allen* nur denkbaren Umgebungsfehlern robust machen können, sondern lediglich für eine Teilmenge von diesen. Diese läßt sich, wie in Abschnitt 4.6.2 beschrieben, durch ein Prädikat Φ_{trap} angeben. Ist jeder

durch Φ_{trap} beschriebene Zustand durch die Erhöhung der Robustheit kein Fangzustand mehr, gilt also

$$\alpha \models \Phi_{trap} \quad \Rightarrow \quad \exists \beta \bullet (\alpha, \beta) \in T \cup F$$

so bleibt das System $\mathcal{S}\triangle(\varnothing, F)$ auch in einer Umgebung reaktiv, in der Eingabeströme erzeugt werden, die eine Teilmenge der abgeschwächten Umgebungsannahme $\mathcal{A}(\mathcal{S}) \cup LeadsTo(\Phi_{trap})$ bilden.

In einer realen, praktischen Systementwicklung will man die Verbesserung der Robustheit eines Systems nicht auf semantischer Ebene, sondern in der Beschreibungstechnik durchführen, in der das System spezifiziert ist. Besonders gut lassen sich Fangzustände in der graphischen Repräsentation von Transitionssystemen erkennen. Jeder durch einen Knoten dargestellte Kontrollzustand s hat eine (n-elementige) Menge von abgehenden Transitionen τ_i, die jeweils mit einer Vorbedingung Φ_i (mit $i \in \{1, \ldots, n\}$) assoziiert sind. Diese Vorbedingungen enthalten sowohl Aussagen über den lokalen Datenzustand als auch über die aktuellen, lesebereiten Nachrichten auf den Eingabekanälen, also $free(\Phi_i) \subseteq I \cup A$. Das System ist *in einem Kontrollzustand s und einer Variablenbelegung α blockiert*, wenn es keine ausgehenden Transitionen gibt (also $n = 0$ gilt) oder wenn $n > 0$ und keine der ausgehenden Transitionen schaltbereit ist, d.h. wenn (mit σ als Parameter für den aktuellen Kontrollzustand) gilt

$$\alpha \quad \models \quad \sigma = s \wedge \neg (\Phi_1 \vee \ldots \vee \Phi_n)$$

Um nachzuweisen, ob ein Kontrollzustand s ein Fangzustand sein kann, muss man diese Aussage aber nicht für alle Belegungen überprüfen. Meist läßt sich ein Kontrollzustand mit gewissen Eigenschaften Ψ verknüpfen, die immer gelten, wenn sich das System im Kontrollzustand s befindet. In diesem Fall stellt $\sigma = s \Rightarrow \Psi$ eine Invariante des Systems dar. Ein Knoten beschreibt damit nicht nur dann einen Fangzustand, wenn die Disjunktion der Vorbedingungen nicht `true` ergibt, sondern auch dann, wenn diese Disjunktion *unter der Voraussetzung* Ψ nicht identisch zu `true` ist. Die Bedingung

$$\Psi \Rightarrow \neg (\Phi_1 \vee \ldots \vee \Phi_n)$$

ist also eine hinreichende Bedingung dafür, dass s ein Fangzustand ist. Da unter Umständen aber Ψ nicht die stärkste Aussage ist, die im Zustand s gilt, ist diese Aussage keine notwendige Bedingung für einen Fangzustand. Die Disjunktion kann für eine Belegung α erfüllbar sein, obwohl diese Belegung im Zustand s in keinem Systemablauf erreichbar ist.

Die Robustheit des Systems kann bei einem vorliegenden Fangzustand s erhöht werden durch eine Transition mit der Vorbedingung Φ_{n+1}, für die gilt

$$\Phi_{n+1} \quad \Rightarrow \quad \sigma = s \ \wedge \ \neg (\Phi_1 \vee \ldots \vee \Phi_n)$$

Damit ist sichergestellt, dass die neue Transition tatsächlich von s ausgeht und nur dann schaltbereit ist, wenn keine der ursprünglichen Transitionen schalten kann. Damit bleibt das Verhalten bei fehlerfreien Eingaben unverändert, wie wir es auch bei Black-Box A/G-Spezifikationen verlangt haben. Die Nachbedingung und der nachfolgende Kontrollzustand der neuen Transition können beliebig definiert werden, und sollten eine sinnvolle Fehlerbehandlung bewirken. Durch iteriertes Ergänzen von Transitionen kann ein System schließlich beliebig robust gegenüber externen Fehlern gemacht werden.

Beispiel 5.7 Unterlauf des Puffers (Transitionssystem)

In Beispiel 4.12 haben wir den Puffer als Transitionssystem spezifiziert und die darin enthaltene implizite Umgebungsannahme abgeleitet. Wir wollen nun erneut wie in Beispiel 5.6 eine Reaktion auf einen Unterlauf spezifizieren. Dazu zeichnen wir den Zustand aus, der einen Unterlauf charakterisiert und ein Fangzustand des Systems ist:

$$\Phi_{trap} \stackrel{df}{=} ft.i^+ = R \;\wedge\; \#q = 0$$

Es liegt in diesem Zustand eine Anfrage vor, es sind aber keine Daten in q gespeichert. Wir definieren für diese Situation wieder eine Reaktion, um die Robustheit gegenüber einem derartigen Umgebungsfehler zu erhöhen: Mit Hilfe der neuen Transition

Name	*Pre*	*Input*	*Output*	*Post*
τ_{uf}	$\#q = 0$	$i?R$	$-$	$q' = q$

wird die nicht sinnvoll beantwortbare Anfrage einfach ignoriert. Die Vorbedingung dieser Transition ist disjunkt zu den Vorbedingungen der bereits vorhandenen Transitionen, so dass das Verhalten im Normalfall unverändert bleibt. □

Die im Rahmen der Erhöhung von Robustheit ergänzten neuen Transitionen verändern die Reaktion des Systems auf Eingaben, für die das System ursprünglich blockiert hat. Durch die Modifikation ist das Systemverhalten verändert worden und entsprechend auch seine Black-Box Eigenschaften. Damit ist ein eventuell bereits geführter Beweis, der die Eigenschaften eines Systems zeigt, nicht mehr gültig. Wir diskutieren im folgenden Abschnitt, wie Beweise an Modifikationen von Systemen und Eigenschaften angepasst werden können.

5.7 Wiederverwendung von Beweisen

Mit den Beweistechniken aus Kapitel 3.8 können wir nachweisen, dass ein Transitionssystem $\mathcal{S}$ gewisse Eigenschaften Φ hat. Werden im Verlauf eines Entwicklungsprozesses potentielle, aber unvermeidbare Fehler eines Systems identifiziert, so können diese mit den in dieser Arbeit präsentierten Techniken als korrespondierende Modifikationen der Eigenschaften bzw. des Transitionssystems dargestellt werden. Ein bereits geführter Beweis für das ursprüngliche und fehlerfreie System macht für das modifizierte, fehlerbehaftete System keine Aussage mehr. Ohne eine Anpassung des Beweises wäre damit der Aufwand, der bereits für die Erstellung des Beweises investiert wurde, verloren. Wir wollen in diesem Abschnitt diskutieren, wie aus dem Beweis für das fehlerfreie System ein Beweis für das fehlerbehaftete System abgeleitet werden kann.

In Abschnitt 3.8.2 haben wir gesehen, wie Beweise für Invarianten und Fortschrittseigenschaften mit Hilfe von Diagrammen repräsentiert werden können. Wir gehen im folgenden von einem Transitionssystem $\mathcal{S}$ aus – zusammen mit einem gültigen Beweisdiagramm für eine Eigenschaft Φ. Sowohl für Invarianten- als auch Fortschrittsdiagramme beschreiben wir nun, ob und wie diese an eine Veränderung der Transitionen angepasst werden können.

5.7.1 Invariantendiagramme

Wir unterscheiden zwischen den Auswirkungen, die das Entfernen und das Hinzufügen von Transitionen auf ein Invariantendiagramm haben.

Entfernen von Transitionen In einem Invariantendiagramm ist mit jedem Knoten und jeder davon ausgehenden Transition eine Beweisverpflichtung verbunden. Damit ist es offensichtlich, dass durch die Entfernung von Transitionen die Zahl der Beweisverpflichtungen reduziert wird. Liegt ein gültiges Beweisdiagramm für ein System $\mathcal{S}$ vor, so stellt dies automatisch auch ein gültiges Diagramm für das System $\mathcal{S}\triangle(E,\varnothing)$ dar.

Es kann möglich sein, dass für das veränderte System mit weniger Transitionen eine stärkere Invariante gezeigt werden kann als die vom ursprünglichen Diagramm bewiesene Aussage

$$\mathcal{S} \Vdash \mathbf{inv}\ \Phi_0 \vee \ldots \vee \Phi_n$$

Folgende Veränderungen am Beweisdiagramm sind zulässig, ohne seine Gültigkeit für das System $\mathcal{S}\triangle(E,\varnothing)$ zu gefährden:

- Eine mit τ beschriftete Kante, die von einem Knoten Φ_j zu einem Knoten Φ_i führt, kann aus dem Diagramm entfernt werden, wenn die Transition τ im modifizierten System nicht mehr enthalten ist, wenn also gilt

  $$\forall\, \alpha, \beta \bullet \alpha \models \Phi_i \wedge \alpha, \beta' \models \tau \quad \Rightarrow \quad (\alpha, \beta) \notin T \setminus E$$

  Wird der Knoten j eines Diagramms durch das Entfernen von Kanten unerreichbar, so kann und muss er aus dem Diagramm gelöscht werden. Das Diagramm beweist dann die stärkere Aussage

  $$\mathcal{S} \Vdash \mathbf{inv}\ \Phi_0 \vee \ldots \vee \Phi_{j-1} \vee \Phi_{j+1} \vee \ldots \vee \Phi_n$$

- Eine Transition τ kann durch E sowohl in seinen Vor- als auch Nachbedingungen so verstärkt werden, dass nicht nur

  $$\Phi_j \wedge \tau \Rightarrow \Phi_i'$$

  gezeigt werden kann, sondern die verstärkte Aussage

  $$\Phi_j \wedge \tau \Rightarrow \Phi_i' \wedge \chi'$$

  mit geeignetem χ. Die Gesamtaussage des Diagramms verstärkt sich dann zu

  $$\mathcal{S} \Vdash \mathbf{inv}\ \Phi_0 \vee \ldots \vee (\Phi_i \wedge \chi) \vee \ldots \vee \Phi_n$$

  Ein Beispiel für eine verstärkte Vorbedingung haben wir in Abschnitt 5.5 kennengelernt: Eine Menge von Transitionen wurde derart eingeschränkt, dass die

darin enthaltenen Transitionen nur noch schalten dürfen, wenn eine Variable xp den Wert false hat. Dieses Wissen darf dann zum Ableiten eines χ verwendet werden. In Beispiel 5.8 werden wir sehen, wie entfernte Transitionen als verstärkte Nachbedingungen formuliert werden können.

Kann das einem Knoten zugeordnete Prädikat verstärkt werden, ist es zudem möglich, auch die Prädikate der Nachfolgeknoten entsprechend zu verstärken, da von $\Phi_i \wedge \Psi$ mehr Aussagen abgeleitet werden können.

Ergänzung von Transitionen Das Ergänzen von Transitionen (aus einer Menge F) erhält im allgemeinen die Gültigkeit eines Diagramms nicht, und kann eine Abschwächung der Invariante nach sich ziehen. Eine systematische Anpassung des Beweisdiagramms erfordert eine Untersuchung des Diagramms für jeden Knoten Φ_j und jeder *neuen* Transition $\tau \in F$. Dabei ist zu überprüfen, wohin diese Transition führt. Folgende drei Fälle sind möglich:

- Ist die Transition nicht schaltbereit, gilt also

  $$\Phi_j \wedge \tau \;\Rightarrow\; \mathsf{false}$$

  so erfordert die Transition in diesem Knoten keine Änderung im Diagramm. Diese Situation tritt typischerweise ein, wenn die Schaltbereitschaft von einem Kontrollzustand abhängt. Hat τ die Aussage $\sigma = s$ als Teil ihrer Vorbedingung, so muss τ nicht in Zusammenhang mit Knoten untersucht werden, deren Prädikat $\sigma \neq s$ impliziert.

- Ist eine Transition schaltbereit, so ist zu prüfen, ob es einen Knoten i gibt, dessen Prädikat Φ_i vom System nach Ausführung der Transition erfüllt wird, also

  $$\Phi_j \wedge \tau \Rightarrow \Phi_i'$$

  Eine entsprechende Kante muss dann im Diagramm ergänzt werden, um die Gültigkeit des Beweisdiagramms sicherzustellen.

- Gibt es keinen Knoten, der den Zustand nach Ausführung der neuen Transition beschreibt, muss das Diagramm um einen neuen Knoten erweitert werden. Dieser ist mit einem Prädikat Φ_{n+1} zu beschriften, für das

  $$\Phi_j \wedge \tau \Rightarrow \Phi_{n+1}'$$

  gilt. Dabei liegt die Schwierigkeit in der geschickten Wahl von Φ_{n+1}, wie wir im folgenden noch erläutern werden.

  Ausgehend von diesem neuen Knoten muss dann für *alle* Transitionen aus $T \cup F$ erneut untersucht werden, in welche Knoten sie jeweils führen. Dabei können noch weitere neue Knoten entstehen.

In Abbildung 5.7 ist eine Illustration eines erweiterten Verifikationsdiagramms dargestellt. Die Disjunktion der Prädikate der m neuen, schattiert dargestellten Knoten stellt die Abschwächung Φ_F der Invariante dar. Es gilt also nur noch

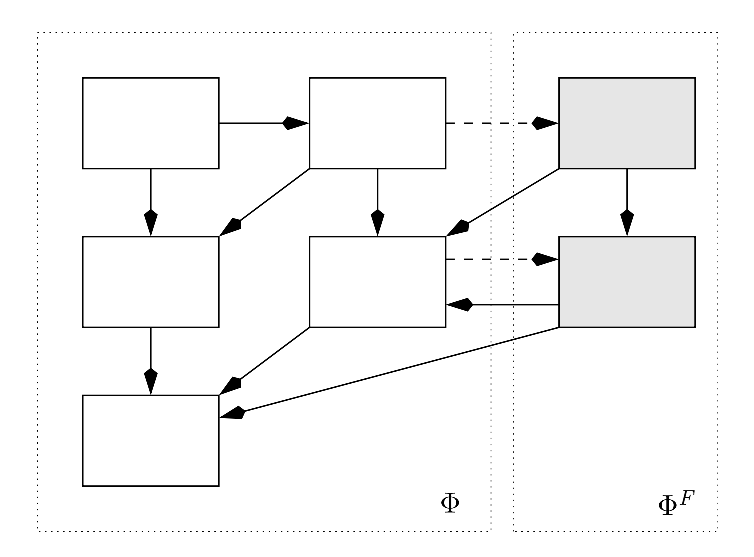


Abbildung 5.7: Erweiterung eines Beweisdiagramms

$$\mathcal{S} \Vdash \textbf{inv} \ \underbrace{\Phi_0 \vee \ldots \vee \Phi_n}_{\Phi} \quad \vee \quad \underbrace{\Phi_{n+1} \vee \ldots \vee \Phi_{n+m}}_{\Phi_F}$$

Das neue Beweisdiagramm umfaßt und verallgemeinert damit das ursprüngliche Beweisdiagramm, so dass damit vom bereits geleisteten Entwicklungsaufwand – auch bei nachträglicher Ergänzung von (Fehler-) Transitionen – möglichst wenig verloren wird.

Die Wahl des Prädikates für einen neuen, einzufügenden Knoten ist dabei aber entscheidend für die Komplexität des neu entstehenden, angepaßten Diagramms. Das Prädikat des neuen Knotens soll einerseits *stark genug* sein, um genügend Wissen über das System zu enthalten und damit sinnvolle Ableitungen möglich zu machen. Das triviale Prädikat `true` wäre ein unbrauchbarer Extremfall, der einen Rücksprung in ein bestehendes Prädikat Φ_i nahezu unmöglich macht, da keine Information über den Systemzustand mehr verfügbar wäre.

Andererseits darf das Prädikat *nicht zu stark* sein, um nicht zu viele weitere Knoten zu generieren. Es sollte eine geeignete Abstraktion des Ausnahmezustandes beschreiben. Wird beispielsweise eine Variable x, die immer nicht-negativ sein sollte, durch einen Fehler dekrementiert, obwohl sie schon den Wert 0 hat, so kann es günstiger sein, diesen Fall als $x < 0$ zu beschreiben und nicht etwas als $x = -1$. Bei einer weiteren Dekrementierung durch die Fehlertransition bleibt dieser Zustand erhalten und es werden nicht weitere generiert mit $x = -2$, $x = -3$, und so weiter. Die geeignete Wahl der Prädikate kann nur mit Kreativität, Intuition und einem Verständnis der Funktionsweise des Systems geschehen, wie dies schon für die Erstellung des ursprünglichen Beweisdiagramms nötig war.

Werden mit einer Modifikation sowohl Transitionen ergänzt als auch entfernt, empfiehlt es sich, bei der Anpassung eines Beweisdiagramms in folgender Reihenfolge vorzugehen:

1. Zunächst werden möglichst viele Kanten und Knoten aus dem Diagramm entfernt.

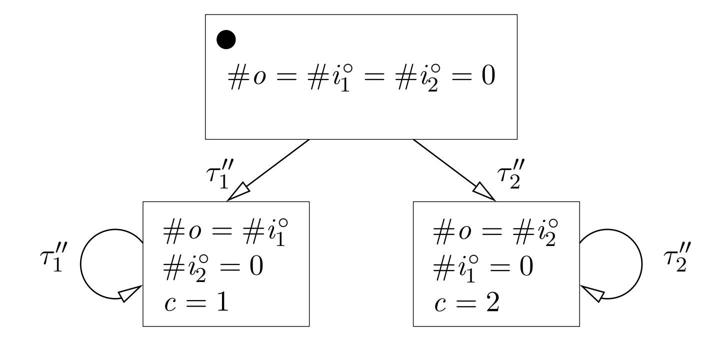


Abbildung 5.8: Modifiziertes Invariantendiagramm für *Merge*

Die Prädikate in den verbleibenden Knoten werden maximal verstärkt.

2. Erst dann werden die neuen Transitionen zusammen mit den notwendigen neuen Knoten hinzugefügt.

Auf diese Weise wird eine unnötig hohe Anzahl von Knoten, Kanten und Beweisverpflichtungen vermieden. Im folgenden Beispiel veranschaulichen wir die Auswirkung von Modifikationen auf ein Beweisdiagramm.

Beispiel 5.8 Anpassung des Invariantendiagramms von *Merge*

In Beispiel 3.11 haben wir mit Hilfe des Invariantendiagramms in Abbildung 3.7 gezeigt, dass

$$\#o = \#i_1^\circ + \#i_2^\circ$$

eine Invariante des Systems *Merge* aus Beispiel 3.6 ist. Wir wollen dieses Beweisdiagramm nun an die in Beispiel 4.5 definierte Modifikation $\mathcal{M} = (E_1 \cup E_2, F)$ anpassen.

Die Entfernung der Transitionen aus $E_1 \cup E_2$ führt zu einer Verstärkung der Vor- und Nachbedingungen von τ_1 und τ_2. Wir nennen die veränderten Transitionen τ_1'' und τ_2'', die wir in Beispiel 4.5 bereits definiert haben als

Name	*Pre*	*Input*	*Output*	*Post*
τ_1''	$c = 1$	$i_1?a$	$o!a$	$c' = c$
τ_2''	$c = 2$	$i_2?a$	$o!a$	$c' = c$

Das Beweisdiagramm 3.7 bleibt mit diesen Transitionen gültig. Wir können die Prädikate aber nun verstärken. Die Transition τ_1'' stellt sicher, dass nach ihrer Ausführung $c = 1$ gilt. Da der linke mittlere Knoten nur mit Hilfe dieser Transition erreicht werden kann, können wir diese Eigenschaft dem Prädikat ergänzen. Analog kann $c = 2$ dem rechten mittleren Knoten hinzugefügt werden.

Mit $c = 1$ im linken Knoten ist es offensichtlich, dass Transition τ_2'' in diesem Zustand nicht mehr schaltbereit ist, da $c = 2$ in der Vorbedingung enthalten ist. Diese Kante kann also entfernt werden. Entsprechend wird die Kante vom rechten Knoten zum unteren Knoten gelöscht. Da der untere Knoten nun nicht mehr erreichbar ist, wird auch er entfernt. Es verbleibt das in Abbildung 5.8 dargestellte Beweisdiagramm.

Wir müssen nun die Auswirkung der neuen Transitionen aus F berücksichtigen. Dazu ergänzen wir zu jedem Knoten die beiden Transitionen

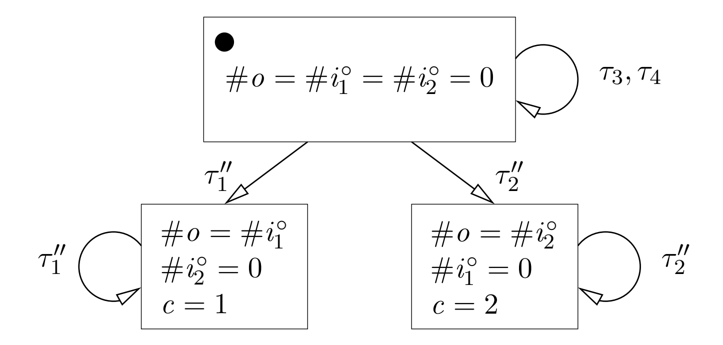


Abbildung 5.9: Vollständig modifiziertes Invariantendiagramm für *Merge*

Name	*Pre*	*Input*	*Output*	*Post*
τ_3	$c \neq 1 \wedge c \neq 2$	–	–	$c = 1$
τ_4	$c \neq 1 \wedge c \neq 2$	–	–	$c = 2$

und prüfen, in welche Knoten deren Ausführung führt. Da beide nur auf den Wert von c Einfluß haben, das im Prädikat des oberen Knoten nicht auftritt, erhalten diese Transitionen offensichtlich die Gültigkeit des Prädikates und können als Schleife eingetragen werden. In den beiden unteren Knoten sind τ_3 und τ_4 nicht schaltbereit und müssen daher auch nicht ergänzt werden.

Als Ergebnis erhalten wir das Beweisdiagramm in Abbildung 5.9. Da keine neuen Knoten hinzugefügt werden mußten, beweist das angepaßte Diagramm noch immer die Invariante. Da ein Knoten entfernt wurde, geht das ihm zugeordnete Prädikat nicht mehr in die Disjunktion ein, so dass mit dem Diagramm sogar

$$\mathit{Merge} \triangle \mathcal{M} \ \Vdash \ \#o = \#i_1^\circ + \#i_2^\circ \wedge (\#i_1^\circ = 0 \vee \#i_2^\circ = 0)$$

gezeigt wird. □

Das Beispiel verdeutlicht, wie das Beweisdiagramm des modifizierten Systems einen nicht unerheblichen Teil des ursprünglichen Beweisdiagramms wiederverwendet. Das neue Diagramm enthält 16 Beweisverpflichtungen (3 Knoten mit jeweils 5 Transitionen und die Gültigkeit des Initialzustandes). Von diesen wurden 10 bereits im alten Diagramm gezeigt, lediglich 6 neue Beweisverpflichtungen entstehen durch die beiden neuen Transitionen.

Selbstverständlich ist der Grad der Wiederverwendung nicht in allen Fällen hoch. Bei starken Modifikationen, die mit vielen neuen Transitionen viele neue Knoten erzeugen, entstehen auch viele neue Beweisverpflichtungen. Dennoch ist es auch in diesem Fall lohnens- und erstrebenswert, auf einem bereits existierenden Beweis aufzubauen, auch wenn dieser unter Umständen nur noch einen Spezialfall des Gesamtbeweises abdeckt.

5.7.2 Fortschrittsdiagramme

Mit einem Fortschrittsdiagramm entsprechend Kapitel 3.8.2 kann eine Eigenschaft der Form $\Phi \mapsto \Psi$ nachgewiesen werden. Wird die Transitionsmenge des Systems verändert, so erfüllt das System im allgemeinen diese Eigenschaft nicht mehr. Wir werden im folgenden untersuchen, in welchen Fällen die Gültigkeit erhalten bleibt oder wie die

Eigenschaft an das veränderte System angepasst werden kann. Dabei unterscheiden wir wieder zwischen dem Entfernen und Ergänzen von Transitionen.

Entfernen von Transitionen Eine mit einem Fortschrittsdiagramm verbundene Beweisverpflichtung fordert, dass von jedem Knoten des Diagramms (außer vom Zielknoten Φ_0) mindestens eine ausgehende Transition schaltbereit ist. Es ist dann sichergestellt, dass eine derartige Transition auch ausgeführt wird. Aufgrund der monoton abnehmenden Bewertungsfunktion nähert sich das System mit jedem Schritt dem Zielzustand Φ_0, der schließlich die Eigenschaft Φ impliziert.

Durch die Entfernung von Transitionen ist es nun möglich, dass eine relevante, zielführende Transition aus dem System entfernt wurde, und das System in seinem Ablauf nicht mehr im Sinne des Fortschrittsdiagramms vorankommt. Dies wird daran erkennbar, dass die Eigenschaft

$$\Phi_j \Rightarrow \mathsf{En}(\tau_{i_1}) \vee \ldots \vee \mathsf{En}(\tau_{i_k})$$

für einen Knoten Φ_j mit den ausgehenden Transitionen $\tau_{i_1}, \ldots, \tau_{i_k}$ nicht mehr gilt. Bleibt diese Eigenschaft für alle Knoten erhalten, so bleibt auch das Fortschrittsdiagramm und damit die Aussage $\Phi \mapsto \Psi$ gültig. Solange es einen Pfad zum Zielknoten Φ_0 gibt, können Kanten gelöscht und unerreichbare Knoten entfernt werden, ohne die Fortschrittsaussage ungültig zu machen.

Ist dies aber nicht der Fall, und will man das Beweisdiagramm nicht vollständig verwerfen, sondern zumindest teilweise beibehalten, so sind folgende Maßnahmen möglich:

- Es werden wiederum Transitionen ergänzt, so dass obige Eigenschaft erneut gilt. Dieser Fall tritt typischerweise ein, wenn man bestehende Transitionen durch ähnliche Transitionen *ersetzt*, die eine andere Wirkung zeigen können, dabei aber die reine Fortschrittseigenschaft nicht beeinträchtigen. Die neuen Transitionen müssen selbstverständlich auf korrekte Weise dem Diagramm hinzugefügt werden, und damit unter anderem sicherstellen, dass das Prädikat des Nachfolgeknotens gilt und durch ihre Ausführung die Bewertung δ monoton verkleinert wird.

- Durch eine Verstärkung von Φ_j zu $\Phi_j \wedge \chi$ werden zusätzliche Annahmen an den Systemzustand im Knoten j gemacht. Es kann möglich sein, mit dieser Verstärkung die geforderte Schaltbereitschaft von einer der verbleibenden Transitionen zu zeigen.

  Dies hat Auswirkungen auf weitere Beweisverpflichtungen, die dann zusätzlich noch gezeigt werden müssen: Alle an Knoten j eingehenden Transitionen müssen zusätzlich die Gültigkeit von χ implizieren. Dies kann wiederum eine Verstärkung der Prädikate ihrer Ausgangsknoten erfordern. In Abschnitt 5.7.1 haben wir die Verstärkung von Prädikaten in Zusammenhang mit der Entfernung von Transitionen bereits beschrieben.

  Da hier zusätzlich noch die Initialbedingung

  $$\Phi \Rightarrow \Phi_0 \vee \ldots \vee (\Phi_j \wedge \chi) \vee \ldots \vee \Phi_n$$

gelten muss, ist unter Umständen sogar Φ zu verstärken.

- Es kann möglich sein, dass die Modifikation das System soweit verändert, dass es im Zustand Φ_j hängenbleibt, und von dort keine Transition mehr in Richtung Φ_0 führt. In einem solchen Fall muss die Fortschrittsaussage abgeschwächt werden zu

  $$\Phi \mapsto \Psi \vee \Phi_j$$

In allen drei Fällen wird durch eine Anpassung des Beweisdiagramms und möglicherweise durch eine Abschwächung der Fortschrittsaussage $\Phi \mapsto \Psi$ ein großer Teil des ursprünglichen Beweisdiagramms wiederverwendet. Die Abschwächung erfolgt dabei durch eine Verstärkung von Φ oder eine Abschwächung von Ψ.

Ergänzung von Transitionen Auch *neue* Transitionen können dazu führen, dass ein Fortschrittsdiagramm nicht mehr gültig ist. Es ist möglich, dass sie in neue Zustände führen, die vom bestehenden Diagramm nicht erfaßt sind, und von denen unter Umständen der Zielknoten nicht mehr erreichbar ist.

Wir gehen wieder von einem bestehenden Diagramm aus, das einen Nachweis für $\Phi \mapsto \Psi$ repräsentiert. Wird das System erweitert um eine Transition τ, muss für jeden Knoten Φ_j untersucht werden, wohin diese Transition von ihm aus führt. Wir unterscheiden wieder drei Fälle, die wir schon von den Invariantendiagrammen kennen:

- Die Transition τ ist im Knoten nicht schaltbereit, also $\Phi_j \wedge \tau \Rightarrow \mathsf{false}$. Die Gültigkeit des Diagramms bleibt erhalten.

- Die Transition führt in einen anderen existierenden Knoten Φ_i bei gleichzeitiger Reduzierung der Bewertung, es gilt also

  $$\Phi_j \wedge \delta_j = u \wedge \tau \quad \Rightarrow \quad \Phi_i' \wedge \delta_i < u$$

  In diesem Fall bleibt weiterhin sichergestellt, dass das System auch nach Ausführung der neuen Transition noch den Zustand Φ_0 erreichen wird.

- Falls die Transition schaltbereit ist, jedoch kein Knoten im Diagramm vorhanden ist, der die notwendigen Bedingungen erfüllt, so ist ein neuer Knoten Φ_{n+1} mit einer Bewertung δ_{n+1} zu ergänzen. Wie schon bei neuen Knoten in Invariantendiagrammen muss bei seiner Formulierung ein angemessener Kompromiß zwischen einer spezifischen, starken und einer allgemeinen, aber dafür schwachen Aussage gewählt werden.

  Jeder neue Knoten muss wieder alle Eigenschaften erfüllen, die in einem Fortschrittsdiagramm von den Knoten gefordert werden. Insbesondere müssen also alle Transitionen wieder in bestehende Knoten führen, und immer muss mindestens eine ausgehende Transition schaltbereit sein. Alternativ ist es auch möglich, die Fortschrittseigenschaft abzuschwächen zu $\Phi \mapsto \Psi \vee \Phi_{n+1}$. Das Ergänzen eines neuen Knotens kann noch weitere neue Knoten erfordern.

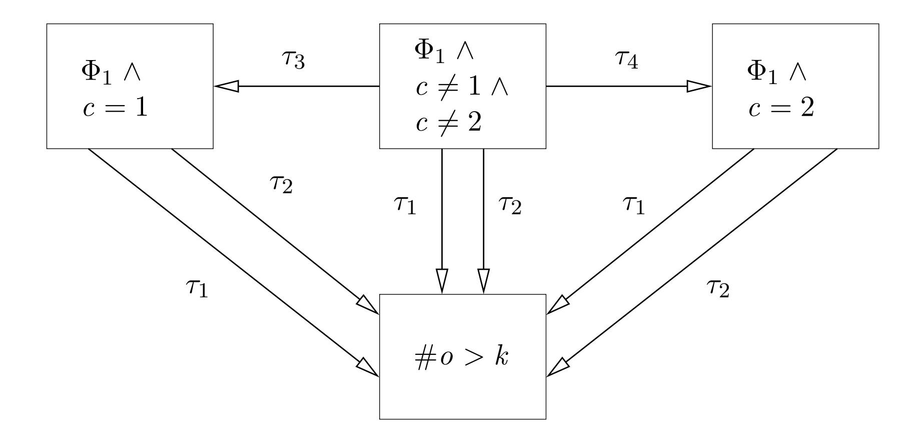


Abbildung 5.10: Erweitertes Fortschrittsdiagramm für *Merge*

Das Erweitern der Transitionsmenge eines Systems korrespondiert mit einer Erweiterung des Fortschrittsdiagramms. Das ursprüngliche Beweisdiagramm kann damit als Teildiagramm vollständig wiederverwendet werden, solange die Fortschrittseigenschaft unverändert erhalten bleibt.

Beispiel 5.9 Anpassung des Fortschrittsdiagramms von *Merge*

Die Komponente *Merge* gibt alle Nachrichten, die ihr über die beiden Eingabekanäle zugesendet werden, auch wieder aus. Diese Fortschrittseigenschaft wurde mit Hilfe des Diagramms in Abbildung 3.8 gezeigt.

Wir wollen nun die Auswirkungen der Modifikation aus Beispiel 4.5 untersuchen. Die Transitionen τ_1 bzw. τ_2 werden dort so eingeschränkt, dass sie nur für $c = 1$ oder für $c = 2$ schaltbereit sind. Im Knoten Φ_1 ist dazu keine Information verfügbar, so dass die erforderliche Schaltbereitschaft in diesem Knoten nicht nachgewiesen werden kann. Wir spalten diesen Knoten auf in drei Knoten, die disjunkten Fällen entsprechen. Das zugehörige Diagramm ist in Abbildung 5.10 dargestellt und hat die identische Aussage wie das Diagramm in 3.8. Dabei beschreibt Φ_1 wieder das Prädikat

$$\Phi_1 \quad \equiv \quad \#o = k \wedge k < \#i_1 + \#i_2$$

Zudem sind bereits die Übergänge der neuen Transitionen von τ_3 und τ_4 eingetragen. Diese legen den Wert von c auf 1 oder 2 fest und sind auch nur genau dann schaltbereit, wenn c noch keinen dieser Werte hat.

Wir modifizieren nun die Transitionen zu τ_1' und τ_2'. Durch die stärkeren Vorbedingungen können einige Kanten gelöscht werden, da τ_1' nur im linken oberen Knoten und τ_2' nur im rechten Knoten schaltbereit ist. Dennoch ist damit kein gültiges Diagramm erreicht, da die Schaltbereitschaft von τ_1' und τ_2' nicht nachgewiesen werden kann. Wir setzen hier lediglich voraus, dass die Anzahl der bisherigen Ausgaben unterhalb der Anzahl der Eingaben an *beiden* Eingabekanälen liegt. Daraus kann jedoch nicht geschlossen werden, dass an einem der beiden Kanäle noch Nachrichten vorhanden sind. Die behauptete Aussage ist für das modifizierte System auch tatsächlich nicht

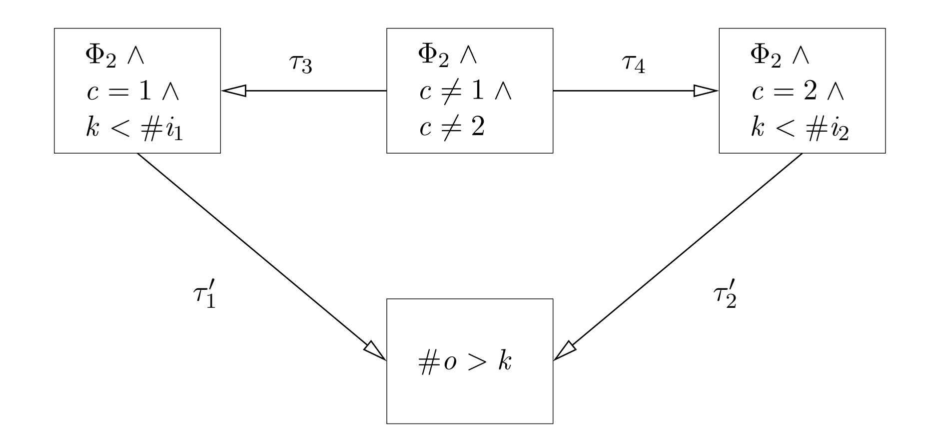


Abbildung 5.11: Modifiziertes Fortschrittsdiagramm für *Merge*

gültig. Wir können sie aber abschwächen[1], und behaupten nur noch

$$\#o = k \wedge k < \#i_1 \wedge k < \#i_2 \quad \mapsto \quad \#o > k$$

Mit Φ_2 definiert als $\#o = k \wedge k < \#i_1 \wedge k < \#i_2$ können wir die Knotenbeschriftungen verstärken und erhalten das Diagramm in Abbildung 5.11.

Als Ausschnitt der darin enthaltenen Beweisverpflichtungen zeigen wir exemplarisch die Schaltbereitschaft von τ_1'. Dazu müssen wir die Eigenschaften aus dem linken oberen Knoten im Diagramm sowie die Invariante verwenden, die wir in Beispiel 5.8 hergeleitet haben, die nach einer weiteren, aber im Beweisdiagramm in Abbildung 5.9 leicht erkennbaren Verstärkung als

$$\#o = \#i_1^\circ + \#i_2^\circ \quad \wedge \quad (\#i_1^\circ = 0 \wedge c = 2) \ \vee \ (\#i_2^\circ = 0 \wedge c = 1))$$

formuliert werden kann. Wir leiten aus $c = 1$ daraus

$$\#o = i_1^\circ + \#i_2^\circ = i_1^\circ$$

ab, so dass wir mit

$$\#o = k < \#i_1 = \#i_1^\circ + \#i_1^+$$

auf

$$\#i_1^+ > 0$$

schließen können. Damit ist τ_1' schaltbereit. □

Beweisdiagramme stellen also nicht nur eine geeignete Methode dar, Beweise zu repräsentieren, sondern ermöglichen es durch ihre graphische und damit intuitive Darstellung auch, die Veränderungen von Eigenschaften bei modifizierten Transitionssystemen zu ermitteln. Es läßt sich gut verfolgen, welche neue Transitionen aus dem Diagramm herausführen bzw. welche entfernten Transitionen die Verstärkung einer Aussage ermöglichen.

[1] Der Einfachheit halber haben wir die Fortschrittseigenschaft mehr als nötig abgeschwächt, und haben nur gezeigt, dass die Ausgabe verlängert wird, wenn $k < min(\#i_1, \#i_2)$ ist. Hat sich das System vermöge der Wahl eines c aber für einen Kanal entschieden, so muss k nur kürzer als der entsprechende Eingabestrom sein.

Allerdings kann die Beweisanpassung im allgemeinen nicht automatisch erfolgen. Bereits die Erstellung eines Diagramms für das unmodifizierte System erfordert ein Verständnis seiner Funktionsweise, so dass auch die Anpassung Einsicht und Kreativität erfordert. Die Anpassung eines Systems und der Beweisdiagramme ist eine komplexe Aufgabe: Fehlertransitionen können teilweise durch weitere hinzuzufügende Modifikationen wieder unschädlich gemacht werden. Diese haben aber auf die beiden Diagrammarten unterschiedliche Auswirkungen. So können beispielsweise durch neue Transitionen Fortschrittseigenschaften einfach wieder gültig gemacht werden, wobei aber wiederum Invarianten ihre Gültigkeit verlieren können.

Das Führen von Beweisen mit Hilfe von Diagrammen ist in einem Entwicklungsprozeß besonders dann hilfreich und gewinnbringend einzusetzen, wenn es durch ein Werkzeug unterstützt wird. In [13] wird dies vorbereitet. Ein Werkzeug sollte es ermöglichen, Beweisdiagramme zu erstellen, die implizierten Beweisverpflichtungen zu generieren und diese nach Möglichkeit sogar zu beweisen.

Die in diesem Ansatz vorgeschlagenen Anpassungen von Beweisdiagrammen könnten gut durch ein Werkzeug unterstützt werden, das die Auswirkungen einer Modifikation ermittelt und bereits gezeigte und neue entstehende Beweisziele verwaltet. Die Beweisverpflichtungen sind in den meisten Fällen zwar einfach, können aber in großer Zahl auftreten. Ohne Werkzeugunterstützung besteht die Gefahr, dass hier einzelne Beweisverpflichtungen übersehen werden oder aufgrund vermeintlicher Ähnlichkeiten und Symmetrien inkorrekte Analogieschlüsse gezogen werden.

Auch in [51] wird der Ansatz diskutiert, Beweise an veränderte Systeme anzupassen, um damit Fehlertoleranz nachzuweisen. Anhand eines Beispiels wird demonstriert, wie große Teile eines ursprünglichen Beweises für den Beweis des veränderten, fehlerbehafteten Systems wiederverwendet werden können. Die Autoren beschränken sich allerdings auf reine Sicherheitsaussagen und auf ein relativ einfaches Fehlermodell. Auch sie bewerten die Anpassung von Beweisen als ein interessantes und offenes Forschungsgebiet. Einen weiteren Ansatz zur Wiederverwendung von Beweisen beschreibt [54].

5.8 Zusammenfassung und Diskussion

Die Einführung formaler Begriffe zur Beschreibung von Modifikationen von Systemen mit einer Untersuchung ihrer Eigenschaften ist als Selbstzweck nicht ausreichend. Sie müssen darüberhinaus in einem Entwicklungsprozeß sinnvoll eingesetzt werden können. In diesem Kapitel haben wir gezeigt, wie typische Entwicklungsschritte für Systeme, in denen Fehler eine Rolle spielen, in unserem Formalismus durchgeführt werden können.

Wir haben dazu einige typische Fehlerklassen formuliert, von denen Systeme betroffen sein können. Damit konnte die allgemeine Ausdruckskraft der Modifikationen gut untermauert werden. Für den praktischen Einsatz ist ein Katalog von Fehlerklassen denkbar, der sinnvollerweise in Kombination mit den jeweils entsprechenden Gegenmaßnahmen zur Verfügung gestellt wird.

Fehlerannahmen in einem System können nicht-lokal sein, wenn die Zusammenhänge und Häufigkeiten des Auftretens von Fehlern mehrere Komponenten betreffen. Wir

haben gezeigt, wie sich diese Zusammenhänge mit bereits existierenden Beschreibungsmitteln unseres Formalismus' geeignet beschreiben lassen. Es können *virtuelle* Komponenten verwendet werden, die nicht implementiert werden, aber eine komponentenübergreifende Annahme über Fehler beschreiben und in einer formalen Untersuchung in der gleichen Weise wie reale Komponenten behandelt werden können.

Wir haben gezeigt, dass die Fehlererkennung mit Hilfe von Zustandsprädikaten formalisiert werden kann, und haben Kriterien angegeben, die ein fehlererkennendes Prädikat relativ zu einem definierten Fehler zu erfüllen hat. Wir sind damit also in der Lage, einen Fehler mit Hilfe von Modifikationen exakt zu definieren, und formal zu überprüfen, ob die Fehlererkennung eines Systems in Bezug auf diesen Fehler adäquat ist.

Als weiteren typischen Schritt in der Entwicklung eines fehlerbehafteten Systems haben wir sowohl die Einführung von (Varianten von) Fehlermeldungen angegeben als auch eine schematische Erweiterung um eine allgemeine Fehlerbehandlung. Dabei haben wir den Stopp eines Systems (zur Sicherstellung eines *fail-safe* Verhaltens) und den Neustart eines Systems als Spezialfälle modelliert. Alle eingeführten Änderungen eines Systems konnten wieder als Modifikationen eines Transitionssystems oder als Modifikationskomponenten dargestellt werden, so dass sich der bereits eingeführte Formalismus erneut als ausreichend mächtig erweist.

Um ein System auch in einer Umgebung verwendbar zu machen, in der (zu definierende) Fehler auftreten, kann dieses auf systematische Weise robuster gemacht werden. Es weist in einer gutartigen Umgebung das ursprünglich spezifizierte Verhalten auf, zeigt aber auch bei auftretenden Fehlern sinnvolle Reaktionen, die in Abhängigkeit vom jeweiligen Anwendungskontext zu definieren sind. Wir haben formale Kriterien für eine Systemmodifikation vorgestellt, die eine derartige Erhöhung der Robustheit sicherstellen.

In einer formalen Systementwicklung im Rahmen unseres Modells kann man Black-Box Eigenschaften eines Transitionssystems verifizieren. Wird ein Transitionssystem durch eine Modifikation verändert, so ändern sich auch seine Eigenschaften. Eine wesentliche Voraussetzung für den Umgang mit Modifikationen von bestehenden, verifizierten Systemen ist also eine Vorgehensweise, mit der auch die entsprechenden Beweise angepasst werden können. Dazu haben wir Techniken vorgestellt, wie durch Diagramme repräsentierte Beweise an Modifikationen von Transitionssystemen und entsprechenden Modifikationen ihrer Eigenschaften angepasst werden können. Mit den Ergebnissen aus 4.5.2 können damit Zusammenhänge von Modifikationen im gesamten Entwicklungsbaum entsprechend Abbildung 3.11 hergestellt werden.

Die in diesem Kapitel vorgestellten methodischen Hinweise sind größtenteils abstrakt formuliert, um möglichst unabhängig von konkreten Beschreibungstechniken die eigentlichen, zugrundeliegenden Anforderungen auszudrücken. Beispielsweise sind die Kriterien eines fehlererkennenden Prädikates über Ausführungen und die darin enthaltenen Zustände formuliert. Es ist wünschenswert, auch für ein konkretes System, das beispielsweise durch Programmcode beschrieben ist, angepaßte Beschreibungstechniken zu finden und die in diesem Kapitel definierten Kriterien entsprechend umzusetzen. Manche Modifikationen lassen sich in konkreten Programmiersprachen sogar trivial umsetzen: Während beispielsweise nach Einführung einer neuen Variable unter Umständen alle

Transitionen entfernt werden müssen, die diesen Wert ändern, ist dies im Ausführungsmodell einer Programmiersprache automatisch sichergestellt. Solange der Wert einer Variable nicht explizit verändert wird, bleibt er gleich.

Wir haben uns in dieser grundlagenorientierten Arbeit auf die Bereitstellung von allgemeinen Prinzipien und die Darstellung von elementaren Zusammenhängen fokussiert. Für die Umsetzung unserer Ansätze in die praktische Verwendung in einer Systementwicklung sind noch weitere Überlegungen anzustellen. Im Vordergrund steht hierbei die Bereitstellung eines Werkzeugs, das einen Entwickler in der Anwendung der Methodik unterstützt. Ein derartiges Werkzeug sollte es möglich machen, Systeme zu beschreiben, Modifikationen darauf anzuwenden und deren Auswirkungen zu untersuchen. So sollte es idealerweise ermöglichen, aus einem Katalog von Fehlermodellen eine Fehlerklasse zu wählen, diese einem System zuzuordnen, gegebenenfalls Fehlererkennungs- und Korrekturmechanismen vorzuschlagen und sogar Beweise anzupassen. Eine Realisierung dieses idealisierten Werkzeugs erfordert aber noch die Lösung einiger Probleme, die wir im nächsten Kapitel zusammenfassen. Vielversprechend sind Ansätze, die beispielsweise Zustände identifizieren, für die die Reaktivität nicht sichergestellt ist oder die die offenen Beweisverpflichtungen in einem modifizierten Beweisdiagramm ermitteln.

Kapitel 6

Zusammenfassung und Ausblick

In diesem abschließenden Kapitel fassen wir die Ergebnisse und Beiträge der vorliegenden Arbeit zusammen und identifizieren die neu aufgeworfenen Problemstellungen mit den daraus resultierenden weiterführenden Aufgaben und Herausforderungen.

6.1 Beitrag dieser Arbeit

Der Kernbeitrag dieser Arbeit liegt in der Integration des Fehlerbegriffs in die formale Methodik Focus. Damit wird eine engere Verbindung geschaffen zwischen der formalen Welt, in der idealisierend von der Abwesenheit jeglicher Fehler ausgegangen wird, und einer realen Welt, in der vielfältige Arten von Fehlern oft unvermeidbar sind und damit auch im Entwicklungsprozeß für komplexe verteilte Softwaresysteme nicht ignoriert werden können und dürfen.

Formale Methoden werden oft gerade deshalb eingesetzt, um Fehlerfreiheit in einer Systementwicklung zu erreichen. Durch die mit ihrer Verwendung verbundene hohe Präzision in der Formulierung von Systemspezifikationen und einer formal nachweisbaren Korrektheit von Entwicklungsschritten werden Fehler im Entwicklungsprozeß weitgehend ausgeschlossen. Jedes konkret realisierte System stützt sich aber ab auf eine nicht notwendigerweise fehlerfreie zugrundeliegende Basis, wie beispielsweise das Betriebssystem, die Hardware oder die Systemumgebung.

Diese Fehler dürfen nicht vernachlässigt werden. Eine Verschiebung des Umgangs mit Fehlern in späte Phasen einer Systementwicklung ist nicht in jedem Fall eine angemessene Lösung. So können Fehler in Einzelkomponenten nicht immer auf lokaler Ebene maskiert werden, sondern setzen sich in ihrer Wirkung in andere Komponenten und schließlich im Gesamtsystem durch. Sie müssen dann auch im Rahmen einer abstrakten, systemglobalen Sicht berücksichtigt werden, und entsprechend bei einer formalen Beschreibung der Systeme mit erfaßt werden. Die Aufnahme und Berücksichtigung von Fehlern stellt damit einen wichtigen Schritt dar von einer idealisierten, fehlerfreien Welt hin zu einer realen Welt mit potentiellen Fehlern. Auch dieser Schritt muss im Rahmen einer formalen Systementwicklung mit unterstützt werden.

Mit den in dieser Arbeit präsentierten Modellierungs- und Entwicklungstechniken liegen alle wesentlichen Grundlagen hierfür vor. Der Fehlerbegriff und die damit zusammenhängenden Konzepte aus seinem thematischen Umfeld wurden formal fundiert und zusammen mit methodischen Hinweisen in die Methodik Focus eingebettet.

Wir geben hier einen Überblick über die wichtigsten Beiträge dieser Arbeit mit ihren jeweiligen Konsequenzen.

- Um sinnvolle und adäquate Fehlerbegriffe zu definieren, haben wir in Kapitel 2 die in der Literatur auffindbaren Interpretationen dieses Begriffes diskutiert und verschiedene Klassifikationen dargestellt. Es ließen sich die Begriffe der *Fehlerursache*, des *Fehlerzustandes* und des *Versagens* als die drei wesentlichen Begriffsfelder identifizieren. Wir haben sie weitgehend im Einklang mit der in der Fachliteratur aufzufindenden Terminologie zunächst informell charakterisiert.

  Ein Überblick über die relevanten Ansätze und den Stand der Technik zur formalen Fehlermodellierung macht Defizite in diesem Gebiet deutlich: Oben genannte Fehlerbegriffe sind nicht oder nur unzulänglich formal definiert, und die Ansätze sind in ihrer Ausdruckskraft eingeschränkt. Sie bieten wenig methodische Unterstützung eines Entwicklungsprozesses und keine pragmatischen Beschreibungstechniken zur Spezifikation fehlerbehafteter Systeme.

  Damit motiviert sich der allgemeinere und umfassendere formale Ansatz, der mit dieser Arbeit verfolgt wurde.

- Als Grundlage für einen präzisen und formalen Umgang mit Fehlern haben wir die Methodik Focus ausgewählt. Sie bietet alle Voraussetzungen, die zur Entwicklung der wichtigen Systemklasse der verteilten, reaktiven Systeme notwendig sind. Die verschiedenen Beschreibungstechniken, die kompositionale Semantik, die mächtigen Verfeinerungsbegriffe, die vorhandenen Beweistechniken und ein bereits verfügbares Werkzeug machen Focus zu einem guten Ausgangspunkt, um es zusammen mit den hier präsentierten Erweiterungen auch in der Praxis einzusetzen.

  Um diese Arbeit auch Lesern ohne Vorkenntnisse über die Methodik Focus zugänglich zu machen, haben wir in Kapitel 3 alle Aspekte von Focus, die für diese Arbeit relevant sind, kurz vorgestellt und anhand von Beispielen veranschaulicht.

- Zur Formalisierung von Fehlern haben wir in Kapitel 4 die sogenannten *Modifikationen* $\mathcal{M}$ eingeführt. Mit einer Modifikation wird der Unterschied zwischen einem System S und einer modifizierten Version $S \triangle \mathcal{M}$ des Systems beschrieben.

  Mit S als einer Beschreibung des *Soll*-Systems und $S \triangle \mathcal{M}$ als *Ist*-System ist die Modifikation $\mathcal{M}$ eine formale Charakterisierung eines *Fehlers*, der eine Abweichung eines Soll vom Ist darstellt. Damit eignen sich Modifikationen hervorragend zur Formalisierung von Fehlern.

  Zur Darstellung konkreter Modifikationen haben wir drei verschiedene Beschreibungstechniken vorgestellt: Die Modifikationen von Black-Box Spezifikationen (Abschnitt 4.2.2), die Modifikationen von Transitionssystemen (Abschnitt 4.2.3)

und Modifikationskomponenten (Abschnitt 4.2.4). Damit können Fehler auf verschiedenen Abstraktionsniveaus beschrieben werden.

Im Gegensatz zu fehlermodellierenden Techniken aus anderen formalen Ansätzen können unsere Modifikationen sowohl Verstärkungen von Eigenschaften als auch Abschwächungen bzw. die Entfernung als auch das Ergänzen von Transitionen beschreiben. Sie sind damit in ihrer Ausdruckskraft sehr allgemein und können nicht nur für Fehler, sondern auch zur Formalisierung intendierter Änderungen wie für *Updates* oder *Patches* von Systemen verwendet werden.

- Den Modifikationsbegriff verwenden wir in Kapitel 4.3 als Grundlage zur formalen Definition der Begriffe *Fehler*, *Fehlerzustand* und *Versagen*. Damit ist die in der Literatur sonst immer nur informell gehaltene Terminologie um präzise Charakterisierungen ergänzt. Diese Definitionen ermöglichen es, die Korrektheit verschiedener Mechanismen nachzuweisen, die für den Umgang mit Fehlern relevant sind.

- Modifikationen erlauben eine formale Definition von *Fehlertoleranz*. Weist ein System unter dem Einfluß von Fehlern eine noch akzeptable Veränderung in seinem Verhalten auf, so nennen wir es fehlertolerant. Mit den in dieser Arbeit vorgestellten Ausdrucksmitteln ist es möglich, sowohl die Fehler als auch die Änderung im Verhalten präzise zu formulieren (Abschnitt 4.4.1). Damit sind explizite und exakte Fehlertoleranzaussagen möglich und sogar verifizierbar.

  Eine alternative Interpretation versteht Fehlertoleranz als die Fähigkeit eines Systems, die Auswirkungen von Fehlern, die in einem Teilsystem oder der Umgebung auftreten, nur in möglichst abgeschwächter Form auf das Gesamtsystem Einfluß nehmen zu lassen. Dazu haben wir in Abschnitt 4.4.3 eine qualitative Klassifikation angegeben, mit der Systeme bezüglich dieser Fähigkeit bewertet werden können.

- Das zugrundeliegende Systemmodell bietet durch seinen klaren Schnittstellenbegriff eine Abgrenzung eines Systems von seiner Umgebung. Damit ist es uns im Gegensatz zu anderen Ansätzen im Bereich der Fehlermodellierung möglich, auch *externe* Fehler eines Systems als solche zu charakterisieren und zu beschreiben. Sowohl für Black-Box Spezifikationen als auch für Transitionssysteme haben wir in Kapitel 4.6 gezeigt, wie externe Fehler angegeben werden können. Darüber hinaus haben wir gezeigt, dass in einem Transitionssystem oft implizite Umgebungsannahmen enthalten sind, die formal abgeleitet werden können. Wir haben dies an einem Beispiel demonstriert.

- In Kapitel 4.5 haben wir Eigenschaften der Modifikationen ermittelt und präsentiert, die in einer Systementwicklung gewinnbringend genutzt werden können. So haben wir die *Kombination* von Modifikationen definiert, mit der die erneute Modifikation von bereits modifizierten Systemen kompakt ausgedrückt werden kann. Mit Hilfe der *Modifikationsfortpflanzung* können in einem zusammengesetzten System aus der Modifikation einer Teilkomponente die Auswirkungen auf das Verhalten des Gesamtsystems konstruktiv ermittelt werden.

Diese Eigenschaften wurden für Modifikationen in den beiden Beschreibungstechniken der Black-Box Spezifikationen und der Transitionssysteme präsentiert.

- Mit der Technik der Verifikationsdiagramme können Beweise von Black-Box Eigenschaften von Transitionssystemen repräsentiert werden. Werden die Eigenschaften oder das Transitionssystem durch Modifikationen verändert, wird ein bestehendes Verifikationsdiagramm im allgemeinen ungültig. Werden aber *korrespondierende Modifikationen* gewählt, so dass das veränderte System die veränderten Eigenschaften aufweist, so ist man an einer passenden Veränderung des Beweisdiagrammes interessiert.

  Wir haben in Kapitel 5.7 sowohl für Invarianten- als auch für Fortschrittsdiagramme Techniken erarbeitet, mit Hilfe derer die Diagramme bei Modifikationen der Eigenschaften oder Systeme entsprechend angepaßt werden können. Ein nicht unerheblicher Anteil der Diagramme bleibt dabei unangetastet, so dass die zugehörigen Beweisverpflichtungen übernommen und nicht neu nachgewiesen werden müssen.

- Eine Systementwicklung läßt sich im wesentlichen als ein Baum darstellen mit einer abstrakten Systemspezifikation als Wurzel, verfeinerten Teilkomponenten als Zwischenknoten und einer implementierungsnahen Beschreibung der Komponenten als Blätter. Im Laufe eines Entwicklungsprozesses kann es sich nun ergeben, dass ein Knoten im Baum modifiziert werden muss. Dies kann beispielsweise durch vorherzusehende Fehler oder veränderte Anforderungen notwendig werden. Mit den Techniken, die uns mit dieser Arbeit vorliegen, ist es möglich, ausgehend vom modifizierten Knoten den Baum schrittweise in alle erforderlichen Richtungen nachträglich anzupassen bis wieder ein korrekter Entwicklungsbaum vorliegt.

  Damit können Fehler oder andere Veränderungen in einem System auch *nachträglich* in einer Systementwicklung berücksichtigt werden, ohne die Entwicklung dabei vollkommen neu beginnen zu müssen. Die Änderungen werden an allen Knoten auf verschiedenen Abstraktionsstufen als Modifikationen repräsentiert.

- Die Verwendbarkeit des Formalismus wird nachgewiesen, indem er in Abschnitt 5.1 exemplarisch in typischen Anwendungsmustern dargestellt wird, wie sie bei der Entwicklung eines Systems mit potentiellen Fehlern auftreten.

  Mit Hilfe von Orakelströmen und virtuellen Komponenten aus Abschnitt 5.2 wird es ermöglicht, *Fehlerannahmen* zu formulieren, die mehrere Komponenten betreffen. Diese Annahmen treten somit explizit und gut dokumentiert als gewöhnliche Teile des Systems auf. Sie müssen nicht, wie in anderen Formalismen oft notwendig, außerhalb des formalen Rahmens erfaßt werden.

  In den Kapiteln 5.3, 5.4 und 5.5 werden Techniken präsentiert, mit denen Fehler *erkannt*, *gemeldet* und *behandelt* werden können. Es wurden Kriterien vorgestellt, die formalisieren, wann eine Fehlererkennung relativ zu einem gegebenen Fehlermodell korrekt ist. Wir haben Modifikationen erarbeitet, die – angewandt auf ein System – dazu führen, dass auf einen erkannten Fehler mit einer Fehlermeldung reagiert wird. Als Verallgemeinerung haben wir eine Modifikation angegeben, die eine frei definierbare Reaktion auf einen erkannten Fehler in ein System einfügt.

Als konkrete Beispiele haben wir den Stopp oder Neustart als Fehlerreaktion eines Systems vorgestellt.

- Oft sind mit einem System Umgebungsannahmen verbunden, die die Systemumgebung zu erfüllen hat, damit das System korrekt funktioniert. Möchte man das System so weiterentwickeln, dass es sich auch in einer Umgebung sinnvoll verhält, die nicht mehr alle Annahmen erfüllt, muss man seine *Robustheit erhöhen*.

  Für einen entsprechenden Entwicklungsschritt haben wir in Kapitel 5.6 Techniken und Korrektheitskriterien für verschiedene Arten von Beschreibungstechniken bereitgestellt. Damit kann ein System auf formal unterstützte Weise um Reaktionen auf externe Fehler erweitert werden.

- Alle vorgestellten Techniken und Konzepte dieser Arbeit wurden durch eine *Vielzahl von Beispielen* veranschaulicht. Damit werden für den Leser die Definitionen und ihre Verwendung illustriert und alle Resultate verdeutlicht.

Die Methodik FOCUS ist als Resultat der vorliegenden Arbeit um einen expliziten Fehlerbegriff mit zugehörigen Entwicklungstechniken erweitert worden, mit dem Systeme und ihre Fehler auf systematische und integrierte Weise modelliert, analysiert und behandelt werden können. Die in Kapitel 1.4 genannten Ziele wurden also erreicht.

Allerdings sind für die Techniken und Konzepte bislang im wesentlichen gerade die grundlegenden Fundamente gelegt worden, die einen gewinnbringenden Einsatz in einer praktischen Systementwicklung zunächst vorbereiten. Für einen unmittelbaren Einsatz in einer praktischen Systementwicklung sind weitere Schritte empfehlenswert, die wir zusammen mit offenen Fragestellungen im nächsten und abschließenden Abschnitt dieser Arbeit zusammenstellen.

6.2 Weiterführende Themengebiete

Wir identifizieren und beschreiben nun die neu aufgeworfenen Aufgabenfelder, die sich zur Fortführung dieser Arbeit anbieten. Wir können dabei zwischen zwei Ausrichtungen unterscheiden: Einerseits bieten sich einige *theoretische* und grundlagenorientiere Erweiterungen an, die das Anwendungsfeld der formalen Fehlermodellierung noch weiter vergrößern. Andererseits ergeben sich Aufgabenstellungen, die auf eine Verbesserung der *praktischen* Verwendbarkeit der vorgestellten Ansätze abzielen. Wir skizzieren zunächst die eher theoretischen Themengebiete:

- Wir haben in dieser Arbeit Systeme auf der Basis des asynchronen und ungezeiteten Modells dargestellt. Dies bietet eine abstrakte Sicht auf Systeme, die aber insbesondere die zeitlichen Aspekte in Systemabläufen nicht reflektiert. Zeit kann im Zusammenhang mit potentiellen Fehlern aber durchaus eine relevante Rolle spielen, da Fehler beispielsweise durch Nachrichten erzeugt werden, die zu spät oder zu früh übertragen werden. Fehler können die Performanz von Systemen beeinflussen, wenn durch die Ausführung einer Fehlerbehandlung die normalen Aktivitäten unterbrochen werden müssen.

Die bereits erarbeiteten Techniken zur Fehlermodellierung und -behandlung sind also auf ein mächtigeres Systemmodell zu verallgemeinern. Dazu gehören neben den getakteten, zeitbehafteten und synchronen Modellen auch Erweiterungen in Richtung mobiler und dynamischer Systeme. Die damit verbundenen weiteren Freiheitsgrade im Verhalten von Systemen bergen ein weiteres Fehlerpotential, das zu modellieren und zu untersuchen ist.

- Mit der formalen Einführung des Begriffs der Modifikationen wurden Fehler explizit als eigene formale Größe definiert, die mit Hilfe verschiedener Beschreibungstechniken und auf unterschiedlichen Abstraktionsstufen dargestellt werden kann. Wir haben bereits einige Eigenschaften dieses Begriffs untersucht und Operatoren für Modifikationen definiert. Eine weiterführende Untersuchung und die Definition zusätzlicher Operatoren ist sinnvoll, da die Ausdruckskraft und damit der Nutzen des Formalismus weiter erhöht werden kann. Wir geben einige potentielle und anspruchsvolle, aber lohnenswerte Weiterentwicklungen an:
  - Eine *Ordnungsrelation* zwischen Modifikationen kann den Vergleich von Fehlern ermöglichen und damit ausdrücken, wann ein Fehler „schlimmer“ ist als ein anderer. Damit werden erwünschte Monotonieaussagen möglich: Toleriert ein System einen Fehler, so toleriert es auch jede schwächere Form dieses Fehlers.
  - Als allgemeinerer Ansatz bietet es sich an, eine *Metrik* für Modifikationen zu definieren. Eine Metrik für Fehler ordnet zwei Fehlermodellen eine Maßzahl zu, die den Unterschied zwischen diesen bewertet. Eine derartige Metrik ermöglicht eine Quantifizierung des Schweregrades von Fehlern.
  - Mit einer derartigen Metrik kann der qualitative Begriff zu *Fehlertoleranz* definiert werden, der als *Dämpfung* verstanden wird: Ein System ist fehlertolerant, wenn die Wirkung, die ein Fehler einer Systemkomponente auf ein System zeigt, einen geringeren Schweregrad hat als der Schweregrad des verursachenden Fehlers.
  - In engem Zusammenhang mit einer Metrik über Fehlern steht ein Begriff, der die *Ähnlichkeit* von Verhalten ausdrückt. Zwei Verhalten sind ähnlich, wenn der als Modifikation ausgedrückte Unterschied angemessen klein ist. Eigenschaften wie *graceful degradation* können damit explizit und präzise dargestellt werden.
  - Es ist wünschenswert, Aussagen über Systeme aus den Aussagen über ihre Komponenten abzuleiten. In Abschnitt 4.5.2 haben wir bereits einige hierfür relevante Zusammenhänge dargestellt. Es ist lohnenswert, weitere *Beweisregeln* zu finden, um komplexere derartige Ableitungen möglich zu machen. So ist es beispielsweise interessant, aus beobachteten Fehlern des Gesamtsystems auf die potentiellen Fehlerursachen innerhalb der Komponenten des Systems Rückschlüsse ziehen zu können. Zusätzliche Beweisregeln werden es ermöglichen, mit Modifikationen sozusagen „rechnen“ zu können.
  - Ein weiteres interessantes Ziel für weiterführende Untersuchungen sind Kriterien und Hinweise zum *konstruktiven* Ermitteln korrespondierender Modifikationen. Führt man Änderungen an einem Transitionssystem durch, so

möchte man die damit verbundenen Änderungen der Systemeigenschaften ableiten. Selbst wenn eine allgemeine und in allen Fällen anwendbare Technik kaum zu finden sein wird, ist eine Einschränkung auf Teilklassen von Systemen vielversprechend. So ist etwa vorstellbar, aus bestimmten Klassen von Modifikationen an Transitionssystemen entsprechende Modifikationen an den Systemeigenschaften konstruktiv ermitteln zu können. Beispiele sind die Fehlerklassen aus Abschnitt 5.1, mit denen sich ihnen entsprechende Änderungen an den Systemeigenschaften assoziieren lassen. Entsprechend lohnenswert ist die Entwicklung von Kriterien und Hinweisen, ob und wie *Beweisdiagramme* möglichst konstruktiv an die Modifikationen von Systemen angepaßt werden können.

- In engem Zusammenhang mit dem Fehlerbegriff steht das Themengebiet des *Testens*. Ein Systemtest hat das Ziel, Fehler in einem System aufzudecken. Mit dem in dieser Arbeit erarbeiteten formalen Fehlerbegriff wird auch eine formale Charakterisierung von Tests möglich. Damit ist man beispielsweise in der Lage, aus definierten Fehlern eines Systems einen zugehörigen Test zu konstruieren, der diese Fehler aufdecken wird. Mit Hilfe einer geeigneten Erweiterung unseres Formalismus wird man so in die Lage versetzt, die Korrektheit eines Tests explizit zu formulieren und auch zu verifizieren. Es ist also lohnenswert, in dieser Richtung weiterführende Untersuchungen durchzuführen und den Testbegriff auch im formalen Umfeld zu etablieren.

Um die *pragmatische* Verwendbarkeit der Ansätze dieser Arbeit in einer realen Systementwicklung noch stärker voranzutreiben, sind unter anderem die folgenden Arbeitfelder sinnvoll:

- In Kapitel 5 haben wir typische Schritte formal charakterisiert, die im Umfeld einer Entwicklung eines fehlerbehafteten Systems auftreten. Die Darstellung ist in großen Teilen allerdings noch sehr semantiknah: So werden beispielsweise Mengen von Transitionen mit Hilfe von Belegungen definiert, die wiederum durch Zustandsprädikate beschrieben sind. Die Charakterisierung eines korrekten Fehlererkennungsprädikates stützt sich sogar auf Systemausführungen ab.

  Für die praktische Verwendung in einer Systementwicklung sollten diese formalen Definitionen nach Möglichkeit aber verborgen und stattdessen auf der Ebene der Beschreibungstechniken ausgedrückt werden. Beispielsweise ist die Ausnahmebehandlung aus Definition 5.1 im Kontext einer konkreten Programmiersprache durch Prozeduren und entsprechende Aufrufe realisierbar, unter Umständen sogar unter Ausnutzung von in einer Sprache bereits implementierten *exception*-Mechanismen.

  Die in dieser Arbeit vorgestellten Techniken stellen damit eine allgemeine und prinzipielle Grundlage zum Umgang mit Fehlern dar, die für die Anwendung in einer konkreten Systementwicklungsumgebung aber noch zu instantiieren sind. Hier können also noch einige sinnvolle Weiterentwicklungen erfolgen.

- Eine durchgängige und effiziente Entwicklung eines komplexen Systems erfordert ein systematisches Vorgehen, das durch sogenannte *Vorgehensmodelle* unterstützt wird. Wir haben mit der vorliegenden Arbeit die Möglichkeiten formaler Methodiken erweitert, da auch Fehler und ihre Auswirkungen in verschiedenen Phasen einer Systementwicklung reflektiert werden können. Die in Kapitel 5 behandelten Themen liefern erste systematische Hinweise, wie mit verschiedenen typischen, fehlerbezogenen Aufgaben in einer Systementwicklung umgegangen werden kann.

  Allerdings liegt damit noch kein umfassendes Vorgehensmodell vor, sondern nur einige seiner Bausteine, deren Kombination zu konkreten, den Gesamtentwicklungsprozeß umfassenden Handlungsanweisungen noch zu leisten ist. Es verbleibt damit weiterhin als lohnende Aufgabe, bestehende Vorgehensmodelle zu analysieren, mit Hilfe von Focus auf eine formale Grundlage zu stellen und schließlich um Aspekte der Fehlermodellierung im Sinne dieser Arbeit zu erweitern.

  Auch die Bereitstellung von *design patterns* stellt einen vielversprechenden Ansatzpunkt dar, den Umgang mit Fehlern in einer Systementwicklung zu unterstützen. In [24] finden sich dazu erste Ansätze.

- Die Akzeptanz einer formalen Methodik in der praktischen, industriellen Anwendung hängt in starkem Maße von der Unterstützung durch ein *Werkzeug* ab. Ein derartiges Werkzeug sollte einen Entwicklungsprozeß durchgängig unterstützen, indem es beispielsweise die Nutzung verschiedenster Beschreibungstechniken anbietet, Konsistenzüberprüfungen durchführen kann, Verfeinerungsschritte unterstützt und auch bei der Beweisführung durch die Erzeugung und idealerweise sogar Verifikation von Beweisverpflichtungen entscheidende Hilfen anbietet. Mit AutoFocus [6] liegt bereits ein Werkzeug für unser Systemmodell vor, allerdings noch ohne die Integration der Fehlermodellierung.

  Dieses Werkzeug ist im Sinne dieser Arbeit durch die Unterstützung des expliziten Umgangs mit Modifikationen als eigenständiges Konzept so zu erweitern, dass damit zusätzlich die Modellierung von Fehlern auf den verschiedensten Abstraktionsebenen und die Integration von Techniken zur Fehlerbehandlung möglich gemacht wird.

Mit der vorliegenden Arbeit sind die Grundlagen für eine formale Fehlermodellierung für verteilte reaktive Systeme zur Verfügung gestellt worden. Für einen pragmatischen und unmittelbaren Einsatz der Techniken empfehlen sich die weiterführenden Arbeiten, die in diesem Abschnitt aufgeführt wurden.

Literaturverzeichnis

[1] Alpern, Bowen and Fred B. Schneider: *Defining Liveness.* Information Processing Letters, 21:181–185, 1985.

[2] Arora, Anish: *A foundation of fault-tolerant computing.* PhD thesis, University of Texas at Austin, 1992.

[3] Arora, Anish and Mohamed Gouda: *Closure and convergence: A foundation of fault-tolerant computing.* IEEE Transactions on Software Engineering, 1993.

[4] Arora, Anish and Sandeep Kulkarni: *Component based design of multitolerance.* IEEE Transactions on Software Engineering, 1998.

[5] Arora, Anish and Sandeep Kulkarni: *Detectors and correctors: A theory of fault-tolerance components.* IEEE Transactions on Software Engineering, 1999.

[6] *Web-Seite des Werkzeuges* AutoFocus. http://autofocus.in.tum.de/.

[7] Bishop, Peter: *Software fault tolerance by design diversity.* In [49], 1995. pages 211 - 229.

[8] Breitling, Max: *Modellierung und Beschreibung von Soll-/Ist-Abweichungen.* In [63], 1999. Seiten 35-44.

[9] Breitling, Max: *Modeling faults of distributed, reactive systems.* In Joseph, Mathai (editor): *FTRTFT 2000 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 1926 in *Lecture Notes in Computer Science*, pages 58–69. Springer, 2000.

[10] Breitling, Max und Jan Philipps: *Black Box Views of State Machines.* Technischer Bericht TUM-I9916, Technische Universität München, Dezember 1999.

[11] Breitling, Max und Jan Philipps: *Diagrams for Dataflow.* In: Grabowski, Jens und Stefan Heymer (Herausgeber): *FBT 2000 - Formale Beschreibungstechniken für verteilte Systeme, 10. GI/ITG Fachgespräch*, Seiten 101–110. Shaker Verlag, Juni 2000.

[12] Breitling, Max and Jan Philipps: *Step by step to histories.* In Rus, Teodor (editor): *AMAST 2000 - Algebraic Methodology And Software Technology*, number 1816 in *Lecture Notes in Computer Science*, pages 11–25. Springer, 2000.

[13] BREITLING, MAX und JAN PHILIPPS: *Verification Diagrams for Dataflow Properties.* Technischer Bericht TUM-I0005, Institut für Informatik, Technische Universität München, 2000.

[14] BROWNE, I. A., Z. MANNA, and H. B. SIPMA: *Generalized temporal verification diagrams.* In *Foundations of Software Technology & Theoretical Computer Science*, number 1026 in *Lecture Notes in Computer Science*, pages 484–498, 1995.

[15] BROY, MANFRED: *Interaction refinement – the easy way.* In BROY, M. (editor): *Program Design Calculi. Springer NATO ASI Series, Series F: Computer and System Sciences, Vol. 118*, 1993.

[16] BROY, MANFRED: *A Functional Rephrasing of the Assumption/Commitment Specification Style.* Technischer Bericht TUM-I9417, Institut für Informatik, Technische Universität München, 1994.

[17] BROY, MANFRED: *Compositional refinement of interactive systems.* Journal of the ACM, 44(5), September 1997.

[18] BROY, MANFRED: *The Specification of System Components by State Transition Diagrams.* Technischer Bericht TUM-I9729, Institut für Informatik, Technische Universität München, 1997.

[19] BROY, MANFRED, FRANK DEDERICHS, CLAUS DENDORFER, MAX FUCHS, THOMAS F. GRITZNER und RAINER WEBER: *The Design of Distributed Systems - An Introduction to* FOCUS. Technischer Bericht SFB 342/2-2/92 A, Technische Universität München, 1993.

[20] BROY, MANFRED und OTTO SPANIOL (Herausgeber): *Informatik und Kommunikationstechnik: VDI Lexikon.* Springer, 1999.

[21] BROY, MANFRED and KETIL STØLEN: *Specification and Development of Interactive Systems - FOCUS on Streams, Interfaces and Refinement.* Monographs in Computer Science. Springer, April 2001.

[22] CRISTIAN, FLAVIU: *A rigorous approach to fault-tolerant computing.* IEEE Trans. Software Engineering, 1985.

[23] CRISTIAN, FLAVIU: *Understanding fault-tolerant distributed systems.* Communication of ACM, 34(2):56–78, December 1991.

[24] DOUGLASS, BRUCE POWEL: *Doing Hard Time – Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns.* The Addison-Wesley object technology series. Addison-Wesley, 1999.

[25] *Web-Seite von* FOCUS. http://www4.in.tum.de/proj/focus/.

[26] *Web-Seite von* Formal Methods Europe. http://www.fmeurope.org/.

[27] GÄRTNER, FELIX C.: *Specifications for Fault Tolerance: A Comedy of Failures.* Technischer Bericht TUD-BS-1998-03, Technische Universität Darmstadt, Oktober 1998.

[28] Gärtner, Felix C.: *Fundamentals of fault-tolerant distributed computing in asynchronous environments.* ACM Computing Surveys, 31(1):1–26, 1999.

[29] Grosu, Radu and Ketil Stølen: *A model for mobile point-to-point data-flow networks without channel sharing.* In Wirsing, Martin and Maurice Nivat (editors): *AMAST 1996 - Algebraic Methodology And Software Technology*, number 1101 in *Lecture Notes in Computer Science.* Springer, 1996.

[30] Grosu, Radu, Ketil Stølen und Manfred Broy: *A Denotational Model for Mobile Point-to-Point Data-flow Networks with Channel Sharing.* Technischer Bericht TUM-I9724, Technische Universität München, 1997.

[31] Herlihy, Maurice and Jeannette Wing: *Specifying graceful degradation in distributed systems.* IEEE Transactions on Parallel and Distributed Systems, 2(1), 1991.

[32] Hinkel, Ursula und Katharina Spies: *Spezifikationsmethodik für mobile, dynamische FOCUS-Netze.* In: Wolisz, A., I. Schieferdecker und A. Rennoch (Herausgeber): *Formale Beschreibungstechniken für verteilte Systeme, GI/ITG-Fachgespräch 1997.* GMD Verlag (St.Augustin), 1997.

[33] Huber, Franz, Bernhard Schätz, Alex Schmidt, and Katharina Spies: *Autofocus — a tool for distributed systems specification.* In *Proceedings FTRTFT'96 — Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 1135 in *Lecture Notes in Computer Science*, 1996.

[34] Huber, Franz, Bernhard Schätz und Katharina Spies: *AutoFocus - Ein Werkzeugkonzept zur Beschreibung verteilter Systeme.* In: Herzog, Ulrich (Herausgeber): *Formale Beschreibungstechniken für verteilte Systeme.* Universität Erlangen-Nürnberg, 1996.

[35] Janowksi, Tomasz: *Fault-tolerant bisimulations and process transformations.* In *FTRTFT'94 – Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 863 in *Lecture Notes in Computer Science.* Springer, 1994.

[36] Janowski, Tomasz: *On bisimulation, fault-monotonicity and provable fault-tolerance.* In *AMAST 97 – International Conference on Algebraic Methodology and Software Technology*, number 1349 in *Lecture Notes in Computer Science.* Springer, 1997.

[37] Janowski, Tomasz: *Semantic and logic for provable fault-tolerance.* Slides for tutorial, Formal Methods Europe, Graz, Austria, 1997.

[38] Janowski, Tomasz and Yun Xiaochun: *Concurrency, faults and atomic transactions: Incremental design for fault-tolerance.* Technical Report 138, The United Nations University UNU, International Institute for Software Technology IIST, August 1998.

[39] Johnson, Barry W.: *Design and Analysis of Fault Tolerant Digital Computing.* Addison-Wesley, 1989.

[40] KINDLER, EKKART: *Sicherheits- und Lebendigkeitseigenschaften: Ein Literaturüberblick*. Technischer Bericht TUM-I9339, Technische Universität München, Dezember 1993. SFB-Bericht Nr. 342/2/93 B.

[41] KULKARNI, SANDEEP and ANISH ARORA: *Compositional design of multitolerant repetitive byzantine agreement*. In *Proceedings of the 18th International Conference on the Foundations of Software Technology and Theoretical Computer Science*, 1997.

[42] KULKARNI, SANDEEP and ANISH ARORA: *Automating the addition of fault-tolerance*. In JOSEPH, MATHAI (editor): *Formal Techniques in Real-Time and Fault-Tolerant Systems*, number 1926 in *Lecture Notes in Computer Science*, pages 82–93. Springer, 2000.

[43] LAPRIE, JEAN-CLAUDE: *Dependability: Basic Concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems*. Springer, 1992.

[44] LEE, PETE and TOM ANDERSON: *Fault Tolerance - Principles and Practice*. Springer, second, revised edition, 1990.

[45] LEVESON, NANCY: *Safeware: System Safety and Computers*. Addison-Wesley, 1995.

[46] LIU, ZHIMING: *Fault-Tolerant Programming by Transformations*. PhD thesis, University of Warwick, July 1991.

[47] LIU, ZHIMING and MATHAI JOSEPH: *A formal framework for fault-tolerant programs*. In *IMA Conference on Mathematics of Dependable Systems*. Oxford Univerity Press, 1993.

[48] LIU, ZHIMING and MATHAI JOSEPH: *Specification and verification of recovery in asynchronous communicating systems*. In [69], 1995. pages 137 - 166.

[49] LYU, MICHAEL R. (editor): *Software Fault Tolerance*. John Wiley & Sons, 1995.

[50] MANNA, ZOHAR and AMIR PNUELI: *Temporal verification diagrams*. In *International Symposium on Theoretical Aspects of Computer Software*, number 789 in *Lecture Notes in Computer Science*, pages 726–765, 1994.

[51] MANTEL, HEIKO und FELIX C. GÄRTNER: *A case study in the mechanical verification of fault tolerance*. Technischer Bericht TUD-BS-1999-08, Institut für Informatik, Technische Universität Darmstadt, November 1999.

[52] PAULSON, LAWRENCE C.: *ML for the Working Programmer*. Cambridge University Press, 1996.

[53] *Web-Seite des Projektes* QUEST. http://www4.in.tum.de/proj/quest/.

[54] REIF, WOLFGANG and KURT STENZEL: *Reuse of proofs in software verification*. In SHYAMASUNDAR, R. (editor): *Foundation of Software Technology and Theoretical Computer Science*, number 761 in *Lecture Notes in Computer Science*, Bombay, India, 1993. Springer.

[55] Renzel, Klaus: *Error Detection.* In: Buschmann, Frank und Dirk Riehle (Herausgeber): *European Pattern Languages of Programming Conference*, Irsee, 1997. Siemens Technischer Bericht 120/SW1/FB.

[56] Rumpe, Bernhard: *Formale Methodik des Entwurfs verteilter objektorientierter Systeme.* Doktorarbeit, Technische Univerität München, 1997.

[57] Rushby, John: *Critical system properties: Survey and taxonomy.* Reliability Engineering and System Safety, 43(2):189–219, 1994.

[58] Rust, Heinrich: *Zuverlässigkeit und Verantwortung: die Ausfallsicherheit von Programmen.* Vieweg, 1994.

[59] Schepers, Henk: *Terminology and paradigms for fault tolerance.* In Vytopil, Jan (editor): *Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 3 – 31. Kluwer Academic Publishers, 1993.

[60] Schneider, Fred B.: *Implementing fault-tolerant services using the state machine approach: A tutorial.* ACM Computing Surveys, 1990.

[61] Schneider, Marco: *Self-stabilization.* ACM Computing Surveys, 1993.

[62] Sommerville, Ian: *Software Engineering.* Addison-Wesley, Fifth edition, 1995.

[63] Spies, Katharina und Bernhard Schätz (Herausgeber): *Formale Beschreibungstechniken für verteilte Systeme. FBT99*, GI/ITG-Fachgespräch, München, 1999.

[64] Stølen, Ketil: *Assumption/Commitment Rules for Data-flow Networks – with an Emphasis on Completeness.* Technischer Bericht TUM-I9516, Institut für Informatik, Technische Universität München, 1995.

[65] Stølen, Ketil, Frank Dederichs und Rainer Weber: *Assumption/Commitment Rules for Networks of Asynchronously Communicating Agents.* Technischer Bericht TUM-I9303, Institut für Informatik, Technische Universität München, 1993.

[66] Storey, Neil: *Safety-critical Computer Systems.* Addison-Wesley, 1996.

[67] Tanenbaum, Andrew S.: *Computer Networks.* Prentice-Hall, 1989.

[68] *Web-Seite der* Virtual Library: Formal Methods. http://archive.comlab.ox.ac.uk/formal-methods/.

[69] Vytopil, Jan (editor): *Formal Techniques in Real-time and Fault-Tolerant Systems.* Kluwer Academic Publishers, 1993.

[70] Wing, Jeannette M.: *Teaching mathematics to software engineers.* Technical Report CMU-CS-95-118R, Carnegie Mellon University, 1995.

Abbildungsverzeichnis

2.1 Zusammengesetztes System mit Umgebung . . . 18
2.2 Vereinfachtes Systemmodell . . . 19
2.3 Merkmale eines Systems . . . 20
2.4 Fehlerbegriffe für Soll-/Ist-Abweichungen . . . 21
2.5 Normal- und Fehlerzustände . . . 27
2.6 Triple Modular Redundancy . . . 31
2.7 Reservekomponenten . . . 32
2.8 Recovery Blocks . . . 33
2.9 N-Versionen-Programmierung . . . 34

3.1 Schnittstelle von *Merge* . . . 41
3.2 Verhalten von *Merge* (Beispiel) . . . 42
3.3 Schnittstelle von *Buffer* . . . 46
3.4 Zustandsübergänge von *Merge* . . . 54
3.5 Komposition von Komponenten . . . 56
3.6 Multiplexer . . . 57
3.7 Invariantendiagramm für *Merge* . . . 64
3.8 Fortschrittsdiagramm für *Merge* . . . 65
3.9 Schnittstellenverfeinerung . . . 68
3.10 Schnittstellenverfeinerung eines komponierten Systems . . . 70
3.11 Idealisierter Entwicklungsprozeß . . . 71

4.1 Verhaltensklassen . . . 76
4.2 Verhaltensmodifikation . . . 83
4.3 Varianten von Modifikationskomponenten . . . 90
4.4 Fehlerzustände in einem Ablauf . . . 95
4.5 Zusammengesetztes System . . . 107

4.6 Korrespondierende Modifikationen 111
4.7 Transitionssystem des Puffers 116

5.1 TMR-System mit virtueller Komponente 126
5.2 Transitionssystem der virtuellen Komponente 127
5.3 Invariantendiagramm für *Buffer* 129
5.4 Varianten korrigierender Transitionen 133
5.5 Kombination von Modifikations- und Treiberkomponente 138
5.6 Fehlerhafter Übertragungskanal mit Treibern 138
5.7 Erweiterung eines Beweisdiagramms 146
5.8 Modifiziertes Invariantendiagramm für *Merge* 147
5.9 Vollständig modifiziertes Invariantendiagramm für *Merge* 148
5.10 Erweitertes Fortschrittsdiagramm für *Merge* 151
5.11 Modifiziertes Fortschrittsdiagramm für *Merge* 152